1. Введение

ВВЕДЕНИЕ

Эта серия статей является руководством по теории и практике разработки синтаксических анализаторов и компиляторов языков программирования. Прежде чем вы закончите чтение этой книги, мы раскроем все аспекты конструирования компиляторов, создадим новый язык программирования, и построим работающий компилятор.

Хотя я по образованию и не специалист в компьютерах, я интересовался компиляторами в течение многих лет. Я покупал и старался разобраться с содержимым практически каждой выпущенной на эту тему книги. И, должен признаться, это был долгий путь. Эти книги написаны для специалистов в компьютерной науке и слишком трудны для понимания большинству из нас. Но с течением лет часть из прочитанного начала доходить до меня. Закрепить полученное позволило то, что я начал самостоятельно пробовать это на своем собственном компьютере. Сейчас я хочу поделиться с вами своими знаниями. После прочтения этой книги вы не станете ни специалистом, ни узнаете всех секретов теории конструирования компиляторов. Я намеренно полностью игнорирую большинство теоретических аспектов этой темы. Вы изучите только практические аспекты, необходимые для создания работающей системы.

В течение всей книги я буду проводить эксперименты на компьютере, а вы будете повторять их за мной и ставить свои собственные эксперименты. Я буду использовать SPF4 и периодически буду включать примеры, написанные в Forth. Эти примеры вы будете копировать себе в компьютер и выполнять. Если у вас не установлен SPF4 вам будет трудно следить за ходом обучения, поэтому я настоятельно рекомендую его поставить. Кроме того, это просто замечательный продукт и для множества других задач!

Некоторые тексты программ будут показаны как примеры или как законченные продукты, которые вы можете копировать без необходимости понимания принципов их работы. Но я надеюсь сделать гораздо больше: я хочу научить вас КАК это делается, чтобы вы могли делать это самостоятельно, и не только повторять то что я делаю но и улучшать.

Такую задачу не решить на одной странице. Я попытаюсь сделать это в нескольких статьях. Каждая статья раскрывает один аспект теории создания компиляторов и может быть изучена в отдельности от всех других. Если вас в настоящее время интересует только какой-то определенный аспект, тогда вы можете обратиться к нужной статье. Каждая статья будет появляться по мере завершения, так что вы должны дождаться последней для того, чтобы считать себя закончившими обучение. Пожалуйста, будьте терпеливы.

В общем, каждая книга по теории создания компиляторов раскрывает множество основ, которые мы не будем рассматривать. Типичная последовательность:

- вступление, в котором описывается что такое компилятор.
- одна или две главы, описывающие задание синтаксиса с использованием формы Бэкуса-Наура (БНФ).
- одна или две главы с описанием лексического анализа, с акцентом на детерминированных и недетерминированных конечных автоматах.
- несколько глав по теории синтаксического анализа, начиная с рекурсивного спуска и заканчивая LALR анализаторами.
- глава, посвященная промежуточным языкам, с акцентом на P-код и обратную польскую запись.
- множество глав об альтернативных путях для поддержки подпрограмм и передачи параметров, описания типов, и т.д.
- завершающая глава по генерации кода, обычно для какого-нибудь воображаемого процессора с простым набором команд.

- финальная глава или две, посвященные оптимизации. Эта глава часто остается непрочитанной, очень часто.

В этой серии я буду использовать совсем другой подход. Для начала, я не остановлюсь долго на выборе. Я покажу вам путь, который работает. Если же вы хотите изучить возможности, хорошо я поддержу вас... но я буду держаться того, что я знаю. Я также пропущу большинство тех теорий, которые заставляют людей засыпать. Не поймите меня неправильно: я не преуменьшаю важность теоретических знаний, они жизненно необходимы, когда приходится иметь дело с более сложными элементами какого либо языка. Но необходимо более важные вещи ставить на первое место. Мы же будем иметь дело с методами, 95% которых не требуют много теории для работы.

Я также буду рассматривать только один метод синтаксического анализа: рекурсивный спуск, который является единственным полностью пригодным методом при ручном написании компилятора. Другие методы полезны только в том случае, если у вас есть инструменты типа Yacc, и вам совсем неважно, сколько памяти будет использовать готовый продукт.

Я также возьму страницу из работы Рона Кейна, автора Small C. Поскольку почти все другие авторы компиляторов исторически использовали промежуточный язык подобно P-коду и разделяли компилятор на две части (front end, который производит P-код, и back end, который обрабатывает P-код, для получения выполняемого объектного кода), Рон показал нам, что очень просто заставить компилятор непосредственно производить выполняемый объектный код в форме языковых утверждений ассемблера. Такой код не самый компактный в мире код... генерация оптимизированного кода - гораздо более трудная работа. Но этот метод работает и работает достаточно хорошо. И чтобы не оставить вас с мыслью, что наш конечный продукт не будет представлять никакой ценности, я собираюсь показать вам как создать компилятор с небольшой оптимизацией.

Наконец, я собираюсь использовать некоторые приемы, которые мне показались наиболее полезными для того, чтобы понимать, что происходит, не продираясь сквозь дремучий лес. Основным из них является использование односимвольных токенов, не содержащих пробелов, на ранней стадии разработки. Я считаю, что если я могу создать синтаксический анализатор для распознавания и обработки I-T-L, то я смогу сделать тоже и с IF-THEN-ELSE. На втором уроке я покажу вам, как легко расширить простой синтаксический анализатор для поддержки токенов произвольной длины. Следующий прием состоит в том что я полностью игнорирую файловый ввод/вывод, показывая этим что если я могу считывать данные с клавиатуры и выводить результат на экран я могу также делать это и с файлами на диске. Опыт показывает, что как только транслятор заработает правильно очень просто перенаправить ввод/вывод на файлы. Последний прием заключается в том, что я не пытаюсь выполнять коррекцию/восстановление после ошибок. Программа, которую мы будем создавать, будет распознавать ошибки и просто остановится на первой из них, точно также как это происходит в Turbo Pascal. Будут и некоторые другие приемы, которые вы увидите по ходу дела. Большинство из них вы не найдете в каком либо учебнике по компиляторам, но они работают.

Несколько слов о стиле программирования и эффективности. Как вы увидите, я стараюсь писать программы в виде маленьких, легко понятных фрагментов. Ни одна из процедур, с которыми мы будем работать, не будет состоять из более чем 15-20 строк. Я горячий приверженец принципа KISS (Keep It Simple, Sidney √ Делай это проще, Сидней) в программировании. Я никогда не пытаюсь сделать что-либо сложное, когда можно сделать просто. Неэффективно? Возможно, но вам понравится результат. Как сказал Брайан Керниган, сначала заставьте программу работать, затем заставьте программу работать быстро. Если позднее вы захотите вернуться и подправить что-либо в вашем продукте, вы сможете сделать это т.к. код будет совершенно понятным. Если вы поступаете так, я, тем не менее, убеждаю вас подождать пока программа не будет выполнять все, что вы от нее хотите.

Я также имею тенденцию не торопиться с созданием модулей до тех пор, пока не обнаружу, что они действительно нужны мне. Попытка предусмотреть все необходимое в будущем может свести вас с ума. В наши время, время экранных редакторов и быстрых компиляторов я буду менять модули тогда, когда почувствую необходимость в более мощном. До тех пор я буду писать только то, что мне нужно.

Заключительный аспект: Один из принципов, который мы будем применять здесь, заключается в том, что мы не будем никого вводить в заблуждение с P-кодом или воображаемыми ЦПУ, но мы начнем с получения работающего, выполнимого объектного кода, по крайней мере, в виде программы на ассемблере. Тем не менее, вам может не понравиться выбранный мной ассемблер┘ это √ ассемблер для микропроцессора 68000, используемый в моей системе (под SK*DOS). Я думаю, что вы найдете, тем не менее, что трансляция для любого другого ЦПУ, например 80x86, совершенно очевидна, так что я не вижу здесь проблемы. Фактически, я надеюсь что кто-то, кто знает язык 8086 лучше, чем я, предоставит нам эквивалент объектного кода.

ОСНОВА

Каждая программа нуждается в некоторых шаблонах┘ подпрограммы ввода/вывода, подпрограммы сообщений об ошибках и т.д. Программы, которые мы будем разрабатывать, не составляют исключения. Я попытался выполнить их на минимальном уровне, чтобы мы могли сконцентрироваться на более важных вещах и не заблудиться. Код, размещенный ниже, представляет собой минимум, необходимый нам, чтобы что-нибудь сделать. Он состоит из нескольких подпрограмм ввода/вывод, подпрограммы обработки ошибок и скелета √ пустой основной программы. Назовем ее Cradle. По мере создания других подпрограмм, мы будем добавлять их к Cradle и добавлять вызовы этих подпрограмм. Скопируйте Cradle и сохраните его, потому что мы будем использовать его неоднократно.

Существует множество различных путей для организации процесса сканирования в синтаксическом анализаторе. В Unix системах авторы обычно используют getc и ungetc. Удачный метод, примененный мной, заключается в использовании одиночного, глобального упреждающего символа. Части процедуры инициализации служит для ╚запуска помпы╩, считывая первый символ из входного потока. Никаких других специальных методов не требуется┘ каждый удачный вызов GetChar считывает следующий символ из потока.

```
- Объявления переменных ----------------------------------------

0 VALUE Look - внешний вид

-- Инструменты --------------------------------------------- ----------

- читать новый символ из потока ввода
: getchar (-) EKEY TO Look ;

- сообщить об ошибке
: error ( c-addr u - ) CR  ^G EMIT  ." Ошибка:" TYPE  ." ." ;

- сообщить об ошибке и остановить
: aborts ( c-addr u - ) error ABORT ;

- сообщить, что ожидалось
: expected ( c-addr u - ) S" Ожидаемые» $+ aborts ;

- проверить  альфа-символ
: alpha? ( char — tf )  >UPC  'A'  'Z'  1+ WITHIN ;

- распознать десятичную цифру
```

```
: dgit? (char - tf) '0' '9' 1+ WITHIN ;

- выводить строку со вкладкой
: emits (c-addr u -) Tab EMIT TYPE ;

- выводить строку с вкладкой и crlf
: emitln (c-addr u -) CR emits ;

- инициализировать
: init (-) getchar;

- Особенности --------------------------------------------- ------

- соответствие определенного входного символа -----------------------------
: match ( char -- )
       DUP Look = IF  DROP getchar
               ELSE  S" `" ROT CHAR-APPEND &' CHAR-APPEND expected
              ENDIF ;

- получить идентификатор -------------------------------------------
: getname ( -- char )
       Look alpha? 0= IF  S" Name" expected  ENDIF
       Look >UPC
       getchar ;

- получить номер ------------------------------------------- -----
: getnum ( -- char )
       Look digit? 0= IF  S" Целое" expected  ENDIF
       Look '0' -
       getchar ;

- Основная программа -------------------------------------------- ----
: cradle (-) init;
```

Скопируйте код, представленный выше, в SPF и откомпилируйте. Удостоверьтесь, что программа откомпилировалась и запустилась корректно. Затем переходим к первому уроку, синтаксическому анализу выражений.

2. Синтаксический анализ выражений

НАЧАЛО

Если вы прочитали введение, то вы уже в курсе дела. Вы также скопировали программу Cradle в Turbo Pascal и откомпилировали ее. Итак, вы готовы.

Целью этой главы является обучение синтаксическому анализу и трансляции математических выражений. В результате мы хотели бы видеть серию команд на ассемблере, выполняющую необходимые действия. Выражение √ правая сторона уравнения, например:

x = 2*y + 3/(4*z)

В самом начале я буду двигаться очень маленькими шагами для того, чтобы начинающие из вас совсем не заблудились. Вы также получите несколько хороших уроков, которые хорошо послужат нам позднее. Для более опытных читателей: потерпите. Скоро мы двинемся вперед.

ОДИНОЧНЫЕ ЦИФРЫ

В соответствии с общей темой этой серии (KISS-принцип, помнишь?), начнем с самого простого случая, который можно себе представить. Это выражение, состоящее из одной цифры.

Перед тем как начать, удостоверьтесь, что у вас есть базовая копия Cradle. Мы будем использовать ее для других экспериментов. Затем добавьте следующие строки:

```
- ---------------------------------------------- -------------
- Разбор и трансляция математического выражения
: expression  ( - ) getnum  (.) S" d#  -> eax mov," $+ emitln ;
- ---------------------------------------------- -------------
```

Теперь запустите программу. Попробуйте ввести любую одиночную цифру. Вы получите результат в виде одной строчки на ассемблере. Затем попробуйте ввести любой другой символ и вы увидите, что синтаксический анализатор правильно сообщает об ошибке.

Поздравляю! Вы только что написали работающий транслятор!

Конечно, я понимаю, что он очень ограничен. Но не отмахивайтесь от него. Этот маленький ╚компилятор╩ в ограниченных масштабах делает точно то же, что делает любой большой компилятор: он корректно распознает допустимые утверждения на входном ╚языке╩, который мы для него определили, и производит корректный, выполнимый ассемблерный код, пригодный для перевода в объектный формат. И, что важно, корректно распознает недопустимые утверждения, выдавая сообщение об ошибке. Кому требовалось больше?

Имеются некоторые другие особенности этой маленькой программы, заслуживающие внимания. Во первых, вы видите, что мы не отделяем генерацию кода от синтаксического анализа┘ как только анализатор узнает что нам нужно, он непосредственно генерирует объектный код. В настоящих компиляторах, конечно, чтение в GetChar должно происходить из файла и затем выполняться запись в другой файл, но этот способ намного проще пока мы экспериментируем.

Также обратите внимание, что выражение должно где-то сохранить результат. Я выбрал регистр D0 процессора 68000. Я мог бы выбрать другой регистр, но в данном случае это имеет смысл.

ВЫРАЖЕНИЯ С ДВУМЯ ЦИФРАМИ

Теперь, давайте немного улучшим то, что у нас есть. По общему признанию, выражение, состоящее только из одного символа, не удовлетворит наших потребностей надолго, так что давайте посмотрим, как мы можем расширить возможности компилятора. Предположим, что мы хотим обрабатывать выражения вида:

1+2

или 4-3

или в общем <term> +/- <term> (это часть формы Бэкуса-Наура или БНФ.)

Для того, чтобы сделать это, нам нужна процедура, распознающая термы и сохраняющая результат, и другая процедура, которая распознает и различает ╚+╩ и ╚┘╩ и генерирует соответствующий код. Но если процедура Expression сохраняет свои результаты в регистре D0, то где процедура Term сохранит свои результаты? Ответ: на том же месте. Мы окажемся перед необходимостью сохранять первый результат процедуры Term где-нибудь, прежде чем мы получим следующий.

В основном, что нам необходимо сделать √ создать процедуру Term, выполняющую то что раннее выполняла процедура Expression. Поэтому просто переименуйте процедуру Expression в Term и наберите новую версию Expression:

```
-- -------------------------------------------------------------
-- Recognize and translate an add
: add ( -- )
   '+' match
   term
   S" ebx -> eax  add," emitln ;
-- -------------------------------------------------------------
-- Recognize and translate a subtract
: subtract ( -- )
   '-' match
   term
   S" ebx -> eax  sub," emitln ;
-- -------------------------------------------------------------
-- Parse and translate an expression
: expression ( -- )
   term
   S" eax -> ebx  mov," emitln
   CASE Look
    '+' OF add       ENDOF
    '-' OF subtract  ENDOF
        S" Addop" expected
   ENDCASE ;
-- -------------------------------------------------------------
```

Когда вы закончите, порядок подпрограмм должен быть следующий:

- Term (старая версия Expression)
- Add
- Subtract
- Expression

(Alternatively, load chap2a.frt).

Теперь запустите программу. Испробуйте любую комбинацию, которую вы только можете придумать, из двух одиночных цифр, разделенных ╚+╝ или ╚-╝. Вы должны получить ряд из четырех инструкций на ассемблере. Затем испытайте выражения с заведомыми ошибками в них. Перехватывает анализатор ошибки?

Посмотрите на полученный объектный код. Можно сделать два замечания. Во первых, сгенерированный код не такой, какой бы написали мы. Последовательность

```
n d# -> eax  mov,
eax  -> ebx  mov,
```

неэффективна. Если бы мы писали этот код вручную, то, возможно, просто загрузили бы данные напрямую в EBX.

Вывод: код, генерируемый нашим синтаксическим анализатором, менее эффективный, чем код, написанный вручную. Привыкните к этому. Это в известной мере относится ко всем компиляторам. Ученые посвятили целые жизни вопросу оптимизации кода и существуют методы, призванные улучшить качество генерируемого кода. Некоторые компиляторы выполняют оптимизацию достаточно хорошо, но за это приходится платить сложностью и в любом случае это проигранная битва┘ возможно никогда не придет время, когда хороший программист на ассемблере не смог бы превзойти компилятор. Прежде чем закончится этот урок, я кратко упомяну некоторые способы, которые мы можем применить для небольшой оптимизации, просто, чтобы показать вам, что мы на самом деле сможем сделать некоторые улучшения без излишних проблем. Но запомните, мы здесь для того, чтобы учиться, а не для того, чтобы узнать насколько компактным мы можем сделать код. А сейчас и на протяжении всей этой серии мы старательно будем игнорировать оптимизацию и сконцентрируемся на получении работающего кода.

Но наш код не работает! В коде есть ошибка! Команда вычитания вычитает D1 (первый аргумент) из D0 (второй аргумент). Но это неправильный способ, так как мы получаем неправильный знак результата. Поэтому исправим процедуру Subtract с помощью замены знака следующим образом:

```
-- ------------------------------------------------------------
-- Recognize and translate a subtract
: subtract ( -- )
   '-' match
   term
   S" ebx -> eax  sub," emitln ;
   S" eax  neg," emitln ;
-- ------------------------------------------------------------
```

Теперь наш код (chap2b.frt) даже еще менее эффективен, но по крайней мере выдает правильный ответ! К сожалению, правила, которые определяют значение математических выражений, требуют, чтобы условия в выражении следовали в неудобном для нас порядке. Опять, это только один из фактов жизни, с которыми вы учитесь жить. Все это возвратится снова, чтобы преследовать нас, когда мы примемся за деление.

Итак, на данном этапе мы имеем синтаксический анализатор, который может распознавать сумму или разность двух цифр. Ранее мы могли распознавать только одиночные цифры. Но настоящие выражения могут иметь любую форму (или бесконечность других). Вернитесь и запустите программу с единственным входным символом ⌠1■.

Не работает? А почему должен работать? Мы только указали анализатору, что единственным правильными видами выражений являются выражения с двумя термами. Мы должны переписать процедуру Expression так, чтобы она была намного более универсальной и с этого начать создание настоящего синтаксического анализатора.

ОБЩАЯ ФОРМА ВЫРАЖЕНИЯ

В реальном мире выражение может состоять из одного или более термов, разделенных "addops" ('+' или '-'). В БНФ это может быть записано как:

<expression> ::= <term> [<addop> <term>]*

Мы можем применить это определение выражения, добавив простой цикл к процедуре Expression:

```
-- -------------------------------------------------------------
-- Parse and translate an expression
: expression ( -- )
   term
   BEGIN   Look '+' =  Look '-' = OR
   WHILE   S" eax -> ebx  mov," emitln
           CASE Look
            '+' OF add      ENDOF
            '-' OF subtract ENDOF
                S" Addop" expected
           ENDCASE
   REPEAT ;
-- -------------------------------------------------------------
```

Эта версия (chap2c.frt) поддерживает любое число термов, и это стоило нам только двух дополнительных строк кода. По мере изучения, вы обнаружите, что это характерно для нисходящих синтаксических анализаторов┘ необходимо только несколько дополнительных строк кода чтобы добавить расширения языка. Это как раз то, что делает наш пошаговый метод возможным. Заметьте также, как хорошо код процедуры Expression соответствует определению БНФ. Это также одна из характеристик метода. Когда вы станете специалистом этого метода, вы сможете превращать БНФ в код синтаксического анализатора примерно с

такой же скоростью, с какой вы можете набирать текст на клавиатуре!

OK, откомпилируйте новую версию анализатора и испытайте его. Как обычно, проверьте что ╚компилятор╩ обрабатывает любое допустимое выражение и выдает осмысленное сообщение об ошибке для запрещенных. Четко, да? Вы можете заметить, что в нашей тестовой версии любое сообщение об ошибке выводится вместе с генерируемым кодом. Но запомните, это только потому, что мы используем экран как ╚выходной файл╩ в этих экспериментах. В рабочей версии вывод будет разделен┘ один в выходной файл, другой на экран.

ИСПОЛЬЗОВАНИЕ СТЕКА

В этом месте я собираюсь нарушить свое правило, что я не представлю что-либо сложное, пока это не будет абсолютно необходимо. Прошло достаточно много времени, чтобы не отметить проблему с генерируемым кодом. В настоящее время синтаксический анализатор использует EAX как «основной» регистр, и EBX для хранения частичной суммы. Эта схема работает отлично потому что мы имеем дело только с "addops" (⌠+■ и ⌠-■) и новое число прибавляется по мере появления. Но в общем форме это не так. Рассмотрим, например выражение

1+(2-(3+(4-5)))

Если мы поместим '1' в , то где мы разместим '2' ? Так как выражение в общей форме может иметь любую степень сложности, то мы очень быстро используем все регистры!

К счастью, есть простое решение. Как и все современные процессоры, x86 имеет стек, который является отличным местом для хранения переменного числа элементов. Поэтому вместо того, чтобы помещать термы в EAX и EBX давайте затолкнем их в стек. Для тех кто незнаком с ассемблером x86 √ помещение в стек пишется как

```
push,
```

и

```
pop,
```

Итак, изменим EmitLn в процедуре Expression на

```
S" eax push," emitln
```

и две строки в Add и Subtract:

```
S" [esp] -> eax  add,  [esp 4 +] -> esp  lea," emitln
```

и

```
S" [esp] -> eax  sub,  [esp 4 +] -> esp  lea," emitln,
```

соответственно. Теперь испытаем компилятор снова и(chap2d.frt) удостоверимся что он работает.

И снова, полученный код менее эффективен, чем был до этого, но это необходимый шаг, как вы увидите.

УМНОЖЕНИЕ И ДЕЛЕНИЕ

Теперь давайте возьмемся за действительно серьезные дела. Как вы знаете, кроме операторов "addops" существуют и другие┘ выражения могут также иметь операторы умножения и деления. Вы также знаете, что существует неявный приоритет операторов или иерархия, связанная с выражениями, чтобы в выражениях типа

2 + 3 * 4,

мы знали, что нужно сначала умножить, а затем сложить. (Видите, зачем нам нужен стек?)

В ранние дни технологии компиляторов, люди использовали различные довольно сложные методы для того чтобы правила приоритета операторов соблюдались. Но, оказывается, все же, что ни один из них нам не нужен┘ эти правила могут быть очень хорошо применены в нашей технике нисходящего синтаксического анализа. До сих пор единственной формой, которую мы применяли для терма была форма одиночной десятичной цифры. В более общей форме мы можем определить терм как произведение показателей (product of factors), то есть

<term> ::= <factor> [<mulop> <factor]*

Что такое показатель? На данный момент это тоже, чем был раннее терм - одиночной цифрой.

Обратите внимание: терм имеет ту же форму, что и выражение. Фактически, мы можем добавить это в наш компилятор осторожно скопировав и переименовав. Но во избежание неразберихи ниже приведен полный листинг всех подпрограмм анализатора. (Заметьте способ, которым мы изменяем порядок операндов в Divide.)

```
-- ---------------------------------------------------------------
-- Parse and translate a math factor
: factor ( -- )
    getnum (.)  S"  d# -> eax mov," $+ emitln ;
-- ---------------------------------------------------------------
-- Recognize and translate a multiply
: multiply  ( -- )
   '*' match
   factor
   S" [esp] dword mul, [esp 4 +] -> esp lea," emitln ;
-- ---------------------------------------------------------------
-- Recognize and translate a divide
: divide ( -- )
   '/' match
   factor
   S" ecx pop, ecx -> eax xchg, eax -> edx mov, #31 b# -> edx sar, ebx idiv,"
emitln ;
-- ---------------------------------------------------------------
-- Parse and translate a math term
: term ( -- )
   factor
   BEGIN   Look '*' =  Look '/' = OR
   WHILE   S" eax push," emitln
           CASE Look
             '*' OF multiply  ENDOF
             '/' OF divide    ENDOF
                    S" Mulop" expected
           ENDCASE
   REPEAT ;
-- ---------------------------------------------------------------
-- Recognize and translate an add
: add ( -- )
   '+' match
   term
   S" [esp] -> eax add, [esp 4 +] -> esp lea," emitln ;
-- ---------------------------------------------------------------
-- Recognize and translate a subtract
: subtract ( -- )
   '-' match
   term
   S" [esp] -> eax sub, [esp 4 +] -> esp lea, eax neg," emitln ;
-- ---------------------------------------------------------------
-- Parse and translate an expression
: expression ( -- )
   term
```

```
   BEGIN   Look '+' =  Look '-' = OR
   WHILE   S" eax push," emitln
           CASE Look
            '+' OF add      ENDOF
            '-' OF subtract ENDOF
                   S" Addop" expected
           ENDCASE
   REPEAT ;
-- ----------------------------------------------------------------
```

Конфетка! Почти работающий транслятор в 52 строк Forth! Получаемый код начинает выглядеть действительно полезным, если не обращать внимание на неэффективность. Запомните, мы не пытаемся создавать сейчас самый компактный код.

КРУГЛЫЕ СКОБКИ

Мы можем закончить эту часть синтаксического анализатора добавив поддержку круглых скобок. Как вы знаете, скобки являются механизмом принудительного изменения приоритета операторов. Так, например, в выражении

2*(3+4) ,

скобки заставляют выполнять сложение перед умножением. Но, что гораздо более важно, скобки дают нам механизм для определения выражений любой степени сложности, как, например

(1+2)/((3+4)+(5-6))

Ключом к встраиванию скобок в наш синтаксический анализатор является понимание того, что не зависимо от того, как сложно выражение, заключенное в скобки, для остальной части мира оно выглядит как простой показатель. Это одна из форм для показателя:

<factor> ::= (<expression>)

Здесь появляется рекурсия. Выражение может содержать показатель, который содержит другое выражение, которое содержит показатель и т.д. до бесконечности.

Сложно это или нет, мы должны позаботиться об этом, добавив несколько строчек в процедуру Factor:

```
-- ----------------------------------------------------------------
-- Parse and translate a math factor

DEFER expression ( -- )

: factor ( -- )
   Look '(' = IF  '(' match  expression  ')' match  EXIT  ENDIF
   getnum (.)  S"  d# -> eax mov," $+ emitln ;
-- ----------------------------------------------------------------
```

Заметьте снова, как легко мы можем дополнять синтаксический анализатор, и как хорошо код Форт соответствует синтаксису БНФ.

Как обычно, откомпилируйте новую версию и убедитесь, что анализатор корректно распознает допустимые предложения и отмечает недопустимые сообщениями об ошибках. (chap2f.frt).

УНАРНЫЙ МИНУС

На данном этапе мы имеем синтаксический анализатор, который поддерживает почти любые выражения, правильно? ОК, тогда испробуйте следующее предложение:

-1

Опс! Он не работает, не правда ли? Процедура Expression ожидает, что все числа будут целыми и спотыкается на знаке минус. Вы найдете, что +3 также не будет работать, так же как и что-нибудь типа:

-(3-2).

Существует пара способов для исправления этой проблемы. Самый легкий (хотя и не обязательно самый лучший) способ √ вставить ноль в начало выражения, так чтобы -3 стал 0-3. Мы можем легко исправить это в существующей версии Expression:

```
-- ------------------------------------------------------------

-- Parse and translate an expression
: expression ( -- )
   Look addop? IF  S" eax -> eax  xor," emitln
             ELSE  term
            ENDIF
   BEGIN Look addop?
   WHILE   S" eax  push," emitln
           CASE Look
            '+' OF add      ENDOF
            '-' OF subtract ENDOF
                  S" Addop" expected
           ENDCASE
   REPEAT ;
-- ---------------------------------------------------------------
```

Я говорил вам, насколько легко мы сможем вносить изменения! На этот раз они стоили нам всего трех новых строчек Паскаля. Обратите внимание на появление ссылки на новую функцию IsAddop. Как только проверка на addop появилась дважды, я решил выделить ее в отдельную функцию. Форма функции IsAddop должна быть аналогична форме функции IsAlpha. Вот она:

```
-- --------------------------------------------------------------
-- Recognize an addop
: addop? ( char -- tf ) DUP '+' =  SWAP '-' =  OR ;
-- --------------------------------------------------------------
```

OK, внесите эти изменения в программу и(chap2g.frt) повторно откомпилируйте. Вы должны также включить IsAddop в базовую копию программы Cradle. Она потребуется нам позже. Сейчас попробуйте снова ввести -1. Вау! Эффективность полученного кода довольно плохая┘ шесть строк кода только для того, чтобы загрузить простую константу┘ но, по крайней мере, правильно работает. Запомните, мы не пытаемся сделать замену Turbo Pascal.

На данном этапе мы почти завершили создание структуры нашего синтаксического анализатора выражений. Эта версия программы должна правильно распознавать и компилировать почти любое выражение, которое вы ей подсунете. Она все еще ограничена тем, что поддерживает показатели состоящие только из одной цифры. Но я надеюсь что теперь вы начинаете понимать, что мы можем расширять возможности синтаксического анализатора делая незначительные изменения. Вы возможно даже не будете удивлены, когда услышите, что переменная или даже вызов функции это просто один из видов показателя.

В следующей главе я покажу, как можно легко расширить наш синтаксический анализатор для поддержки всех этих возможностей, и я также покажу как легко мы можем добавить многосимвольные числа и имена переменных. Итак, вы видите, что мы совсем недалеко от действительно полезного синтаксического анализатора.

СЛОВО ОБ ОПТИМИЗАЦИИ

Раннее в этой главе я обещал дать несколько подсказок как мы можем повысить качество генерируемого кода. Как я сказал, получение компактного кода не является главной целью этой книги. Но вам нужно по крайней мере знать, что мы не зря проводим свое время┘ что мы действительно можем модифицировать анализатор для получения лучшего кода не выбрасывая то, что мы уже сделали к настоящему времени.

Обычно небольшая оптимизация не слишком трудна┘ просто в синтаксический анализатор вставляется дополнительный код.

Существуют два основных метода, которые мы можем использовать:

- Попытаться исправить код после того, как он сгенерирован.
  Это понятие «щелевой» оптимизации. Основная идея в том, что известно какие комбинации инструкций компилятор собирается произвести и также известно которые из них «плохие» (такие как код для числа -1). Итак, все что нужно сделать √ просканировать полученный код, найти такие комбинации инструкций и заменить их на более «хорошие». Это вид макрорасширений наоборот и прямой пример метода сопоставления с образцом. Единственная сложность в том, что может существовать множество таких комбинаций. Этот метод называется «щелевой» оптимизацией просто потому, что оптимизатор работает с маленькой группой инструкций. «Щелевая» оптимизация может драматически влиять на качество кода и не требует при этом больших изменений в структуре компилятора. Но все же за это приходится платить скоростью, размером и сложностью компилятора. Поиск всех комбинаций требует проверки множества условий, каждая из которых является источником ошибки. И, естественно, это требует много времени.
  В классической реализации «щелевого» оптимизатора, оптимизация выполняется как второй проход компилятора. Выходной код записывается на диск и затем оптимизатор считывает и обрабатывает этот файл снова. Фактически, оптимизатор может быть даже отдельной от компилятора программой. Так как оптимизатор только обрабатывает код в маленьком «окне» инструкций (отсюда и название), лучшей реализацией было бы буферизировать несколько срок выходного кода и сканировать буфер каждый раз после EmitLn.
- Попытаться сразу генерировать лучший код.
  В этом методе выполняется проверка дополнительных условий перед выводом кода. Как тривиальный пример, мы должны были бы идентифицировать нуль и выдать CLR вместо загрузки, или даже совсем ничего не делать, как в случае с прибавлением нуля, например. Конкретней, если мы решили распознавать унарный минус в процедуре Factor вместо Expression, то мы должны обрабатывать √1 как обычную константу, а не генерировать ее из положительных. Ни одна из этих вещей не является слишком сложной для реализации просто они требуют включения дополнительных проверок в код, поэтому я не включил их в программу. Как только мы дойдем до получения работающего компилятора, генерирующего полезный выполнимый код, мы всегда сможем вернуться и доработать программу для получения более компактного кода. Именно поэтому в мире существует «Версия 2.0».

Существует еще один, достойный упоминания, способ оптимизации, обещающий достаточно компактный код без излишних хлопот. Это мое «изобретение», в том смысле, что я нигде не видел публикаций по этому методу, хотя я и не питаю иллюзий что это придумано мной.

Способ заключается в том, чтобы избежать частого использования стека, лучше используя регистры центрального процессора. Вспомните, когда мы выполняли только сложение и вычитание, то мы использовали регистры EAX и EBX а не стек? Это работало, потому для этих двух операций стек никогда не использовал более чем две ячейки.

Хорошо, процессор x86 имеет четыре регистра данных. Почему бы не использовать их как стек? В любой момент своей работы синтаксический анализатор «знает» как много элементов в стеке, поэтому он может правильно ими манипулировать. Мы можем определить частный

указатель стека, который следит, на каком уровне мы находимся и адресует соответствующий регистр. Процедура Factor, например, должна загружать данные не в регистр EAX, а в тот, который является текущей вершиной стека.

Что мы получаем заменяя стек в RAM на локальный стек созданный из регистров. Для большинства выражений уровень стека никогда не превысит восьми, поэтому мы получаем достаточно хороший код. Конечно, мы должны предусмотреть те случаи, когда уровень стека превысит восемь, но это также не проблема. Мы просто позволим стеку перетекать в стек ЦПУ. Для уровней выше восьми код не хуже, чем тот, который мы генерируем сейчас, а для уровней ниже восьми он значительно лучше.

Я реализовал этот метод, просто для того, чтобы удостовериться в том, что он работает перед тем, как представить его вам. Он работает. На практике вы не можете в действительности использовать все восемь уровней… вам, как минимум, нужен один свободный регистр для изменения порядка операндов при делении. Для выражений, включающих вызовы функций, также необходимо зарезервировать регистр. Но все равно, существует возможность улучшения размера кода для большинства выражений.

Итак, вы видите, что получение лучшего кода не настолько трудно, но это усложняет наш транслятор… это сложность, без которой мы можем сейчас обойтись. По этой причине, я очень советую продолжать игнорировать вопросы эффективности в этой книге, усвоив, что мы действительно можем повысить качество кода не выбрасывая того, что уже сделано.

В следующей главе я покажу вам как работать с переменными и вызовами функций. Я также покажу вам как легко добавить поддержку многосимвольных токенов и пробелов.

3. Снова выражения

ВВЕДЕНИЕ

В последней главе мы изучили методы, используемые для синтаксического анализа и трансляции математических выражений в общей форме. Мы закончили созданием простого синтаксического анализатора, поддерживающего выражения произвольной сложности с двумя ограничениями:

- Разрешены только числовые показатели
- Числовые показатели ограничены одиночной цифрой.

В этой главе мы избавимся от этих ограничений. Мы также расширим то что сделали, добавив операции присваивания и вызовы функций. Запомните, однако, что второе ограничение было главным образом наложено нами самими… выбрано для удобства, чтобы облегчить себе жизнь и сконцентрироваться на фундаментальных принципах. Как вы увидите, от этого ограничения легко освободиться, так что не слишком задерживайтесь на этом. Мы будем использовать это прием пока он служит нам, уверенные в том, что сможем избавиться от него, когда будем готовы.

ПЕРЕМЕННЫЕ

Большинство выражений, который мы встречаем на практике, включают переменные, например:

b * b + 4 * a * c

Ни один компилятор нельзя считать достаточно хорошим, если он не работает с ними. К счастью, это тоже очень просто сделать.

Не забудьте, что в нашем синтаксическом анализаторе в настоящее время существуют два вида показателей: целочисленные константы и выражения в скобках. В нотации БНФ:

<factor> ::= <number> | (<expression>)

"|" заменяет "или", означая, что любая из этих форм является допустимой. Запомните, также, что у нас нет проблемы в определении каждой их них┘ предсказывающим символом в одном случае является левая скобка "(" и цифра √ в другом.

Возможно, не вызовет слишком большого удивления то, что переменная √ это просто еще один вид показателя. Так что расширим БНФ следующим образом:

<factor> ::= <number> | (<expression>) | <variable>

И снова, здесь нет неоднозначности: если предсказывающий символ √ буква, то это переменная, если цифра то число. Когда мы транслируем число, мы просто генерируем код для загрузки числа, как промежуточных данных, в D0. Сейчас мы делаем то же самое, только для переменной.

Небольшое осложнение при генерации кода возникает из того факта, что большинство операционных систем для 68000, включая SK*DOS которую я использую, требуют чтобы код был написан в "переместимой" форме, что в основном означает что все должно быть PC-относительно. Формат для загрузки на этом языке будет следующим:

MOVE X(PC),D0

где X, конечно, имя переменной. Вооружившись этим, изменим текущую версию процедуры Factor следующим образом:

```
-- ---------------------------------------------------------------------------
-- Parse and translate a math factor

DEFER expression ( -- )

: factor ( -- )
   Look '('  = IF  '(' match  expression ')' match EXIT
               ENDIF
   Look alpha? IF  S" dword-ptr -> eax mov," getname CHAR-PREPEND emitln EXIT
               ENDIF
   S"  d# -> eax mov," getnum '0' + CHAR-PREPEND emitln ;
 -- --------------------------------------------------------------------------
```

Я уже отмечал, как легко добавлять расширения в синтаксический анализатор благодаря способу его структурирования. Вы можете видеть, что это все еще остается действительным и сейчас. На этот раз это стоило нам всего двух дополнительных строк кода. Заметьте так же, как структура if-then-else точно соответствует синтаксическому уравнению БНФ.

OK, откомпилируйте (chap3a.frt)и протестируйте эту новую версию синтаксического анализатора. Это не слишком сильно повредило, не так ли?

ФУНКЦИИ

Есть еще только один распространенный вид показателей, поддерживаемый большинством языков: вызов функции. В действительности, нам пока слишком рано иметь дела с функциями, потому что мы еще не обращались к вопросу передачи параметров. Более того, "настоящий" язык должен включать механизм поддержки более чем одного типа, одним из которых должен быть тип функции. Мы не имеем также и этого. Но все же я хотел бы работать с функциями сейчас по двум причинам. Во-первых, это позволит нам превратить компилятор во что-то очень близкое к конечному виду и, во вторых, это раскроет новую проблему, о которой очень стоит поговорить.

До этого момента мы создавали то, что называется "предсказывающим синтаксическим анализатором". Это означает, что в любой точке мы можем, смотря на текущий предсказывающий символ, точно знать, что будет дальше. Но не в том случае когда мы добавляем функции. В каждом языке имеются некоторые правила присваивания имен, по которым составляется допустимый идентификатор. Наши правила пока просты, так как идентификатором является одна из букв "a"┘"z". Проблема состоит в том, что имена

переменных и имена функций подчиняются одним и тем же правилам. Поэтому как мы можем сказать кто из них кто? Один из способов требует, чтобы каждое из них было объявлено перед тем, как оно используется. Этот метод использует Pascal. Другой способ состоит в том, чтобы функция сопровождалась списком параметров (возможно пустым). Это правило, используемое в C.

Пока у нас нет механизма описания типов, давайте использовать правила C. Так как у нас также нет и механизма для работы с параметрами, мы можем поддерживать только пустые списки параметров, так что вызовы функций будут иметь следующую форму:

X().

Так как мы пока не работаем со списками параметров, для вызова функций не нужно ничего дополнительно, и необходимо только выдавать BSR (вызов) вместо MOVE.

Сейчас существуют две варианта для ветки "If IsAlpha" при проверке в процедуре Factor. Давайте обработаем их в отдельной процедуре. Изменим процедуру Factor следующим образом:

```
-- ----------------------------------------------------------------
-- Parse and translate a math factor

DEFER expression ( -- )

: factor ( -- )
   Look '('  = IF  '(' match  expression  ')' match  EXIT  ENDIF
   Look alpha? IF  ident EXIT  ENDIF
   S"  d# -> eax mov," getnum '0' + CHAR-PREPEND emitln ;
-- ----------------------------------------------------------------
```

и вставим перед ней новое слово

```
-- ----------------------------------------------------------------
-- Parse and translate an identifier
: ident ( -- )
   getname LOCAL name
   Look '(' = IF   '(' match ')' match
                S"  offset NEAR call,"
            ELSE   S"  dword-ptr -> eax mov,"
           ENDIF
        name CHAR-PREPEND  emitln ;
-- ----------------------------------------------------------------
```

Откомпилируйте (chap3b.frt) и протестируйте эту версию. Обрабатывает ли она все правильные выражения и корректно отмечает неправильные?

Важно отметить, что хотя наш анализатор больше не является предсказывающим анализатором, это немного или совсем не добавляет сложностей при использовании нами метода рекурсивного спуска. В том месте, где процедура Factor находит идентификатор (букву), она не знает, является ли он именем переменной или именем функции, ни выполняет ее обработку. Она просто передает его в Ident и оставляет этой процедуре на рассмотрение. Ident, в свою очередь, просто прячет идентификатор и затем считывает еще один символ для того, чтобы решить с каким типом идентификатора он имеет дело.

Запомните этот способ. Это очень мощное понятие и оно должно быть использовано всегда, когда вы встречаетесь с неоднозначной ситуацией, требующей заглядывания вперед. Даже если вам нужно рассмотреть несколько символов вперед, принцип все еще будет работать.

ПОДРОБНЕЕ ОБ ОБРАБОТКЕ ОШИБОК

Имеется еще одна важная проблема, которую стоит отметить: обработка ошибок. Обратите внимание, что хотя синтаксический анализатор правильно отбрасывает (почти)

каждое некорректное выражение, которое мы ему подбросим, со значимым сообщением об ошибке, в действительности мы не слишком много поработали для того, чтобы это происходило. Фактически во всей программе (от Ident до Expression) есть только два вызова подпрограммы обработки ошибок Expected. Но даже они не являются необходимыми┘ если вы посмотрите снова на процедуры Term и Expression, то увидите, что эти утверждения не выполнятся никогда. Я поместил их сюда ранее для небольшой подстраховки, но сейчас они более не нужны. Почему бы не удалить их сейчас?

Но как мы получали такую хорошую обработку ошибок фактически бесплатно? Просто я тщательно старался избежать чтения символа непосредственно используя GetChar. Взамен я возложил на GetName, GetNum и Match выполнение всей обработки ошибок для меня. Проницательные читатели заметят, что некоторые вызовы Match (к примеру в Add и Subtract) также не нужны┘ мы уже знаем чем является символ к этому времени┘ но их присутствие сохраняет некоторую симметрию, и было бы хорошим правилом всегда использовать Match вместо GetChar.

Выше я упомянул "почти". Есть случай, когда наша обработка ошибок оставляет желать лучшего. Пока что мы не сказали нашему синтаксическому анализатору как выглядит конец строки или что делать с вложенными пробелами. Поэтому пробел (или любой другой символ, не являющийся частью признаваемого набора символов) просто вызывает завершение работы анализатора, игнорируя нераспознанные символы.

Можно рассудить, что в данном случае это приемлемое поведение. В "настоящем" компиляторе обычно присутствует еще одно утверждение, следующее после того, с которым мы работаем, так что любой символ, не обработанный как часть нашего выражения, будет или использоваться или отвергаться как часть следующего.

Но это также очень легко исправить, даже если это только временно. Все, что мы должны сделать √ постановить, что выражение должно заканчиваться концом строки, то есть, возвратом каретки.

Чтобы понять о чем я говорю, испробуйте входную строку:

1+2 <space> 3+4

Видите, как пробел был обработан как признак завершения? Чтобы заставить компилятор правильно отмечать это, добавьте строку

```
Look ^M <> IF  S" Newline" expected  ENDIF
```

в основную программу, сразу после вызова Expression. Это отлавливает все левое во входном потоке. Не забудьте определить CR в разделе const:

```
CR = ^M;
```

Как обычно откомпилируйте программу(chap3c.frt) и проверьте, что она делает то, что нужно.

ПРИСВАИВАНИЕ

Итак, к этому моменту мы имеем синтаксический анализатор, работающий очень хорошо. Я хотел бы подчеркнуть, что мы получили это, используя всего 88 строк выполнимого кода, не считая того, что было в Cradle. Откомпилированный объектный файл занял 4752 байта. Неплохо, учитывая то, что мы не слишком старались сохранять размеры как исходного так и объектного кода. Мы просто придерживались принципа KISS.

Конечно, анализ выражений не настолько хорош без возможности что-либо делать с его результатами. Выражения обычно (но не всегда) используются в операциях присваивания в форме:

<Ident> = <Expression>

Мы находимся на расстоянии вздоха от возможности анализировать операции присваивания, так что давайте сделаем этот последний шаг. Сразу после процедуры Expression добавьте следующую новую процедуру:

```
-- -------------------------------------------------------------------------
-- Parse and translate an assignment statement
: assignment ( -- )
   getname LOCAL name
   '=' match
   expression
   S" eax -> " 'OF name 1 $+ S"  dword-ptr  mov," $+ emitln ;
-- -------------------------------------------------------------------------
```

Обратите внимание снова, что код полностью соответствует БНФ. И заметьте затем, что проверка ошибок была безболезненна и обработана GetName и Match.

Необходимость двух строк ассемблера возникает из-за особенности 68000, который требует такого вида конструкции для PC-относительного кода.

Теперь измените вызов Expression в основной программе на Assigment. Это все, что нужно.

Фактически мы компилируем операторы присваивания! Если бы это был единственный вид операторов в языке, все, что нам нужно было бы сделать √ поместить его в цикл и мы имели бы полноценный компилятор!

Конечно, это не единственный вид. Есть также немного таких элементов, как управляющие структуры (ветвления и циклы), процедуры, объявления и т.д. Но не унывайте. Арифметические выражения, с которыми мы имели дело, относятся к самым вызывающим элементам языка. По сравнению с тем, что мы уже сделали, управляющие структуры будут выглядеть простыми. Я расскажу о них в пятой главе. И все другие операторы поместятся в одной строчке, пока мы не забываем принцип KISS.

МНОГОСИМВОЛЬНЫЕ ТОКЕНЫ.

В этой серии я тщательно ограничивал все, что мы делаем, односимвольными токенами, все время уверяя вас, что не составит проблемы расширить их до многосимвольных. Я не знаю, верили вы мне или нет┘ я действительно не обвинил бы вас, если бы вы были немного скептичны. Я буду продолжать использовать этот подход и в следующих главах, потому что это позволит избежать сложности. Но я хотел бы поддержать эту уверенность и показать вам, что это действительно легко сделать. В процессе этого мы также предусмотрим обработку вложенных пробелов. Прежде чем вы сделаете следующие несколько изменений, сохраните текущую версию синтаксического анализатора под другим именем. Я буду использовать ее в следующей главе и мы будем работать с односимвольной версией.

Большинство компиляторов выделяют обработку входного потока в отдельный модуль, называемый лексическим анализатором (сканером). Идея состоит в том, что сканер работает со всей последовательностью символов во входном потоке и возвращает отдельные единицы (лексемы) потока. Возможно придет время, когда мы также захотим сделать что-то вроде этого, но сейчас в этом нет необходимости. Мы можем обрабатывать многосимвольные токены, которые нам нужны, с помощью небольших локальных изменений в GetName и GetNum.

Обычно признаком идентификатора является то, что первый символ должен быть буквой, но остальная часть может быть алфавитно-цифровой (буквы и цифры). Для работы с ними нам нужна другая функция:

```
-- ---------------------------------------------------------------
-- Recognize an alphanumeric
: alnum? ( char -- tf )  DUP alpha? SWAP digit? OR ;
-- ---------------------------------------------------------------
```

Добавьте эту функцию в анализатор. Я поместил ее сразу после IsDigit. Вы можете также включить ее как постоянного члена в Cradle.

Теперь нам необходимо изменить функцию GetName так, чтобы она возвращала строку вместо символа:

```
-- ------------------------------------------------------------
-- Get an identifier

CREATE token 0 C, 256 CHARS ALLOT

: char+! ( c addr -- )  DUP >R COUNT + C!  1 R> C+! ;

: getname ( -- c-addr u )
   Look alpha? 0= IF  S" Name" expected  ENDIF
   token C0!
   BEGIN  Look alnum?
   WHILE  Look >UPC token char+!  getchar
   REPEAT token COUNT ;
-- ------------------------------------------------------------
```

Аналогично измените GetNum следующим образом:

```
-- ------------------------------------------------------------
-- Get a number

CREATE #value  0 C, 256 CHARS ALLOT

: getnum ( -- c-addr u )
   Look digit? 0= IF  S" Integer" expected  ENDIF
   #value C0!
   BEGIN  Look digit?
   WHILE  Look #value char+!  getchar
   REPEAT #value COUNT ;
-- ------------------------------------------------------------
```

Требуется небольшое изменение factor:

```
-- ------------------------------------------------------------
: factor ( -- )
   Look '('  = IF  '(' match  expression  ')' match  EXIT  ENDIF
   Look alpha? IF  ident EXIT  ENDIF
   getnum S"  d# -> eax mov," $+ emitln ;
-- ------------------------------------------------------------
```

Достаточно удивительно, что это фактически все необходимые изменения! Локальная переменная Name в процедурах Ident и Assignment были первоначально объявлены как "char" и теперь должны быть объявлены как string[8]. (Ясно, что мы могли бы сделать длину строки больше, если бы захотели, но большинство ассемблеров в любом случае ограничивают длину.)

iForth does not support LOCAL strings:

```
-- --------------------------------------------------------------------------------
: ident ( -- )
   getname DUP 1+ ALLOCATE ?ALLOCATE LOCAL name
   name PACK DROP
   Look '(' = IF  '(' match ')' match
                  name COUNT S"  offset NEAR call,"
            ELSE  name COUNT S"  dword-ptr -> eax mov,"
           ENDIF  $+ emitln
   name FREE ?ALLOCATE ;
-- --------------------------------------------------------------------------------
: assignment ( -- )
   getname DUP 1+ ALLOCATE ?ALLOCATE LOCAL name
```

```
   name PACK DROP
   '=' match  expression  S" eax -> " name COUNT $+ S"  dword-ptr  mov," $+ emitln
   name FREE ?ALLOCATE ;
-- ---------------------------------------------------------------------------------
```

Внесите эти изменения и затем откомпилируйте (chap3e.frt). и протестируйте. Сейчас вы верите, что это просто?

ПРОБЕЛЫ

Прежде, чем мы оставим этот синтаксический анализатор на некоторое время, давайте обратимся к проблеме пробелов. На данный момент, синтаксический анализатор выразит недовольство (или просто завершит работу) на одиночном символе пробела, вставленном где-нибудь во входном потоке. Это довольно недружелюбное поведение. Так что давайте немного усовершенствуем анализатор, избавившись от этого последнего ограничения.

Ключом к облегчению обработки пробелов является введение простого правила для того, как синтаксический анализатор должен обрабатывать входной поток и использование этого правила везде. До настоящего времени, поскольку пробелы не были разрешены, у нас была возможность знать, что после каждого действия синтаксического анализатора предсказывающий символ Look содержит следующий значимый символ, поэтому мы могли немедленно выполнять его проверку. Наш проект был основан на этом принципе.

Это все еще звучит для меня как хорошее правило, поэтому мы будем его использовать. Это означает, что каждая подпрограмма, которая продвигает входной поток, должна пропустить пробелы и оставить следующий символ (не являющийся пробелом) в Look. К счастью, так как мы были осторожны и использовали GetName, GetNum, и Match для большей части обработки входного потока, только эти три процедуры (плюс Init) необходимо изменить.

Неудивительно, что мы начинаем с еще одной подпрограммы распознавания:

```
-- -------------------------------------------------------------
-- Recognize white space
: white? ( char -- tf )  DUP Tab = SWAP BL = OR ;
-- -------------------------------------------------------------
```

Нам также нужна процедура, "съедающая" символы пробела до тех пор, пока не найдет отличный от пробела символ:

```
-- -------------------------------------------------------------
-- Skip over leading white space
: skipwhite ( -- )
   BEGIN  Look white?
   WHILE  getchar
   REPEAT ;
-- -------------------------------------------------------------
```

Сейчас добавьте вызовы SkipWhite в Match, GetName и GetNum как показано ниже:

```
-- --------------------------------------------------------------
-- Match a specific input character
: match ( char -- )
   DUP Look = IF  DROP getchar skipwhite
            ELSE  S" `" ROT CHAR-APPEND  &' CHAR-APPEND expected
           ENDIF ;
-- --------------------------------------------------------------
-- get an identifier
: getname ( -- c-addr u )
   Look alpha? 0= IF  S" Name" expected  ENDIF
   token C0!
   BEGIN  Look alnum?
   WHILE  Look >UPC token char+!  getchar
   REPEAT token COUNT  skipwhite ;
```

```
-- get a number
: getnum ( -- c-addr u )
   Look digit? 0= IF  S" Integer" expected  ENDIF
   #value C0!
   BEGIN   Look digit?
   WHILE   Look #value char+!  getchar
   REPEAT  #value COUNT  skipwhite ;
-- ---------------------------------------------------------
```

(Обратите внимание, как я немного реорганизовал Match без изменения функциональности.)

Наконец, мы должны пропустить начальные пробелы в том месте, где мы "запускаем помпу" в Init:

```
-- ---------------------------------------------------------
-- Initialize
: init ( -- ) getchar skipwhite ;
-- ---------------------------------------------------------
```

Внесите эти изменения и повторно откомпилируйте программу. Вы обнаружите, что необходимо переместить Match ниже SkipWhite чтобы избежать сообщение об ошибке от компилятора Pascal. Протестируйте программу как всегда,(chap3f.frt) чтобы удостовериться, что она работает правильно.

Поскольку мы сделали довольно много изменений в течение этого урока, ниже я воспроизвожу полный текст синтаксического анализатора:

```
-- Variable declarations
--------------------------------------------------------------------------------

0 VALUE Look  -- lookahead character

CREATE token  PRIVATE  0 C, #256 CHARS ALLOT
CREATE #value PRIVATE  0 C, #256 CHARS ALLOT

-- Tools ------------------------------------------------------------------------

: getchar  ( -- )  EKEY TO Look ;           -- read new character from input stream
: error    ( c-addr u -- ) CR ^G EMIT ." Error: " TYPE ." ." ;   -- report an error
: aborts   ( c-addr u -- ) error ABORT ;                   -- report error and halt
: expected ( c-addr u -- ) S"  Expected" $+ aborts ;    -- report what was expected
: alpha?   ( char -- tf ) >UPC 'A' 'Z' 1+ WITHIN ; -- recognize an alpha character
: digit?   ( char -- tf ) '0' '9' 1+ WITHIN ;         -- recognize a decimal digit
: alnum?   ( char -- tf ) DUP alpha? SWAP digit? OR ; -- recognize alphanumeric
: emits    ( c-addr u -- ) Tab EMIT TYPE ;            -- output a string with tab
: emitln   ( c-addr u -- ) CR emits ;         -- output a string with tab and crlf
: addop?   ( char -- tf )  DUP '+' = SWAP '-' = OR ;        -- test for AddOp
: mulop?   ( char -- tf )  DUP '*' = SWAP '/' = OR ;        -- test for MulOp
: white?   ( char -- tf )  DUP Tab = SWAP BL  = OR ;      -- recognize white space

: char+!   ( c addr -- )   DUP >S COUNT + C!  1 S> C+! ;

-- skip white space
: skipwhite ( -- )
        BEGIN  Look white?
        WHILE  getchar
        REPEAT ;

-- Specifics --------------------------------------------------------------------

-- initialize
: init ( -- ) CR getchar skipwhite ;

-- match a specific input character
: match ( char -- )
```

```
        DUP Look = IF  DROP getchar skipwhite
                 ELSE  S" `" ROT CHAR-APPEND &' CHAR-APPEND expected
                ENDIF ;

-- get an identifier
: getname ( -- c-addr u )
        Look alpha? 0= IF  S" Name" expected  ENDIF
        token C0!
        BEGIN  Look alnum?
        WHILE  Look >UPC token char+!  getchar
        REPEAT token COUNT  skipwhite ;

-- get a number
: getnum ( -- c-addr u )
        Look digit? 0= IF  S" Integer" expected  ENDIF
        #value C0!
        BEGIN  Look digit?
        WHILE  Look #value char+!  getchar
        REPEAT #value COUNT  skipwhite ;

-- parse and translate a math expression
---------------------------------------------------------------------------------

DEFER expression ( -- )

-- parse and translate an identifier.
-- Note that we need to store getname locally because parsing a call's
-- parameterlist may clobber it.
: ident ( -- )
        getname DUP 1+ ALLOCATE ?ALLOCATE LOCAL name
        name PACK DROP
        Look '(' = IF  '(' match ')' match
                       name COUNT S"  offset NEAR  call,"
                 ELSE  name COUNT S"  dword-ptr -> eax  mov,"
                ENDIF  $+ emitln
        name FREE ?ALLOCATE ;

: factor ( -- )
        Look '('  = IF  '(' match  expression  ')' match  EXIT  ENDIF
        Look alpha? IF  ident EXIT  ENDIF
        getnum S"  d# -> eax mov," $+ emitln ;

: multiply ( -- ) '*' match factor S" [esp] dword mul, [esp 4 +] -> esp lea,"
emitln ;
: divide    ( -- ) '/' match factor S" ebx pop, ebx -> eax xchg, eax -> edx mov, #31
b# -> edx sar, ebx idiv," emitln ;

: term ( -- )
        factor
        BEGIN Look mulop?
        WHILE   S" eax push," emitln
                CASE Look
                  '*' OF multiply  ENDOF
                  '/' OF divide    ENDOF
                         S" Mulop" expected
                ENDCASE
        REPEAT ;

: add    ( -- ) '+' match term  S" [esp] -> eax add, [esp 4 +] -> esp lea," emitln ;
: subtract ( -- ) '-' match term  S" [esp] -> eax sub, [esp 4 +] -> esp lea, eax
neg," emitln ;

:NONAME ( -- )
        Look addop? IF  S" eax -> eax xor," emitln
                  ELSE  term
                 ENDIF
        BEGIN   Look addop?
```

```
        WHILE   S" eax push," emitln
                CASE Look
                  '+' OF add      ENDOF
                  '-' OF subtract ENDOF
                         S" Addop" expected
                ENDCASE
        REPEAT ; IS expression

-- Parse and translate an assignment statement.
-- Note that we need to store getname locally because parsing a call's
-- parameterlist may clobber it.
: assignment ( -- )
    getname DUP 1+ ALLOCATE ?ALLOCATE LOCAL name
    name PACK DROP
    '=' match  expression  S" eax -> " name COUNT $+ S"  dword-ptr  mov," $+ emitln
        name FREE ?ALLOCATE ;

-- Main Program ------------------------------------------------------------------

: main ( -- )
        init  assignment
        Look ^M <> IF  S" Newline" expected  ENDIF ;
```

Теперь синтаксический анализатор закончен. Он получил все возможности, которые мы можем разместить в однострочном "компиляторе". Сохраните его в безопасном месте. В следующий раз мы перейдем к новой теме, но мы все рано будем некоторое время говорить о выражениях. В следующей главе я планирую рассказать немного об интерпретаторах в противоположность компиляторам и показать вам как немного изменяется структура синтаксического анализатора в зависимости от изменения характера принимаемых действий. Информация, которую мы рассмотрим, хорошо послужит нам позднее, даже если вы не интересуетесь интерпретаторами. Увидимся в следующий раз.

4. Интерпретаторы

ВВЕДЕНИЕ

В трех первых частях этой серии мы рассмотрели синтаксический анализ и компиляцию математических выражений, постепенно и методично пройдя от очень простых односимвольных "выражений", состоящих из одного терма, через выражения в более общей форме и закончив достаточно полным синтаксическим анализатором, способным анализировать и транслировать операции присваивания с многосимвольными токенами, вложенными пробелами и вызовами функций. Сейчас я собираюсь провести вас сквозь этот процесс еще раз, но уже с целью интерпретации а не компиляции объектного кода.

Если эта серия о компиляторах, то почему мы должны беспокоиться об интерпретаторах? Просто я хочу чтобы вы увидели как изменяется характер синтаксического анализатора при изменении целей. Я также хочу объединить понятия этих двух типов трансляторов, чтобы вы могли видеть не только различия но и сходства.

Рассмотрим следующее присваивание:

x = 2 * y + 3

В компиляторе мы хотим заставить центральный процессор выполнить это присваивание во время выполнения. Сам транслятор не выполняет никаких арифметических операций┘ он только выдает объектный код, который заставит процессор сделать это когда код выполнится. В примере выше компилятор выдал бы код для вычисления значения выражения и сохранения результата в переменной x.

Для интерпретатора, напротив, никакого объектного кода не генерируется. Вместо этого арифметические операции выполняются немедленно как только происходит синтаксический анализ. К примеру, когда синтаксический анализ присваивания завершен, x будет содержать

новое значение.

Метод, который мы применяем во всей этой серии, называется "синтаксически-управляемым переводом". Как вы знаете к настоящему времен, структура синтаксического анализатора очень близко привязана к синтаксису анализируемых нами конструкций. Мы создали процедуры на Pascal, которые распознают каждую конструкцию языка. Каждая из этих конструкций (и процедур) связана с соответствующим "действием", которое выполняет все необходимое как только конструкция распознана. В нашем компиляторе каждое действие включает выдачу объектного кода для выполнения позднее во время исполнения. В интерпретаторе каждое действие включает что-то для немедленного выполнения.

Что я хотел бы, чтобы вы увидели, это то, что план┘ структура┘ синтаксического анализатора не меняется. Изменяются только действия. Так что, если вы можете написать интерпретатор для данного языка, то вы можете также написать и компилятор, и наоборот. Однако, как вы увидите, имеются и отличия, и значительные. Поскольку действия различны, процедуры, завершающие распознавание, пишутся по-разному. Характерно, что в интерпретаторе завершающие подпрограммы распознавания написаны как функции, возвращающие числовое значение вызвавшей их программе. Ни одна из подпрограмм анализа нашего компилятора не делает этого.

Наш компилятор, фактически, это то, что мы могли бы назвать "чистым" компилятором. Как только конструкция распознана, объектный код выдается немедленно. (Это одна из причин, по которым код не очень эффективный.) Интерпретатор, который мы собираемся построить, является чистым интерпретаторов в том смысле, что здесь нет никакой трансляции типа "токенизации", выполняемой над исходным текстом. Это две крайности трансляции. В реальном мире трансляторы не являются такими чистыми, но стремятся использовать часть каждой методики.

Я могу привести несколько примеров. Я уже упомянул один: большинство интерпретаторов, типа Microsoft BASIC, к примеру, транслируют исходный текст (токенизируют его) в промежуточную форму, чтобы было легче выполнять синтаксический анализ в реальном режиме времени.

Другой пример √ ассемблер. Целью ассемблера, конечно, является получение объектного кода и он обычно выполняет это по однозначному принципу: одна инструкция на строку исходного кода. Но почти все ассемблеры также разрешают использовать выражения как параметры. В этом случае выражения всегда являются константами, и ассемблер не предназначен выдавать для них объектный код. Скорее он "интерпретирует" выражение и вычисляет соответствующее значение, которое фактически и выдается с объектным кодом.

Фактически, мы могли бы использовать часть этого сами. Транслятор, который мы создали в предыдущей главе, будет покорно выплевывать объектный код для сложных выражений, даже если каждый терм в выражении будет константой. В этом случае было бы гораздо лучше, если бы транслятор вел себя немного как интерпретатор и просто вычислял соответствующее значение константы.

В теории компиляции существует понятие, называемое "ленивой" трансляцией. Идея состоит в том, что вы не просто выдаете код при каждом действии. Фактически, в крайнем случае вы не выдаете что-либо вообще до тех пор, пока это не будет абсолютно необходимо. Для выполнения этого, действия, связанные с подпрограммами анализа, обычно не просто выдают код. Иногда они это делают, но часто они просто возвращают информацию обратно вызвавшей программе. Вооружившись этой информацией, вызывающая программа может затем сделать лучший выбор того, что делать.

К примеру, для данного выражения

x = x + 3 - 2 - (5 - 4)

наш компилятор будет покорно выплевывать поток из 18 инструкций для загрузки каждого параметра в регистры, выполнения арифметических действий и сохранения результата.

Ленивая оценка распознала бы, что выражение, содержащее константы, может быть рассчитано во время компиляции и уменьшила бы выражение до

x = x + 0

Даже ленивая оценка была бы затем достаточно умной, чтобы понять, что это эквивалентно

x = x,

что совсем не требует никаких действий. Мы смогли уменьшить 18 инструкций до нуля!

Обратите внимание, что нет никакой возможности оптимизировать таким способом наш компилятор, потому что каждое действие выполняется в нем немедленно.

Ленивая оценка выражений может произвести значительно лучший объектный код чем тот который мы могли произвести. Я, тем не менее, предупреждаю вас: это значительно усложняет код синтаксического анализатора, потому что каждая подпрограмма теперь должна принять решение относительно того, выдать объектный код или нет. Ленивая оценка конечно же названа так не потому, что она проще для создателей компиляторов!

Так как мы действуем в основном по принципу KISS, я не буду более углубляться в эту тему. Я только хочу, чтобы вы знали, что вы можете получить некоторую оптимизацию кода, объединяя методы компиляции и интерпретации. В частности Вы должны знать, что подпрограммы синтаксического анализа в более интеллектуальном трансляторе обычно что-то возвращают вызвавшей их программе и иногда сами ожидают этого. Эта главная причина обсуждения интерпретаторов в этой главе.

ИНТЕРПРЕТАТОР

Итак, теперь, когда вы знаете почему мы принялись за все это, давайте начнем. Просто для того, чтобы дать вам практику, мы начнем с пустого Cradle и создадим транслятор заново. На этот раз, конечно, мы сможем двигаться немного быстрее.

Так как сейчас мы собираемся выполнять арифметические действия, то первое, что мы должны сделать √ изменить функцию GetNum, которая до настоящего момента всегда возвращала символ (или строку). Лучше если сейчас она будет возвращать целое число. Сделайте копию Cradle (на всякий случай не изменяйте сам Cradle!!) и модифицируйте GetNum следующим образом:

```
-- ---------------------------------------------------------------
-- Get a number
: getnum ( -- n )
   Look digit? 0= IF  S" Integer" expected  ENDIF
   Look '0' -
   getchar ;
-- ---------------------------------------------------------------
```

Затем напишите следующую версию Expression:

```
-- ---------------------------------------------------------------
-- Read new character from input stream and echo it.
: echo     ( char -- ) DUP ^M = IF ^J EMIT ENDIF  EMIT ;
: getchar  ( -- )      EKEY DUP echo TO Look ;
-- ---------------------------------------------------------------
```

И, наконец, вставьте

```
-- ---------------------------------------------------------------

-- Parse and translate an expression
```

```
: expression ( -- n ) getnum ;
-- ------------------------------------
```

в конец основной программы.

```
CR expression .
```

Теперь откомпилируйте и протестируйте.

Все, что эта программа делает - это "анализ" и трансляция "выражения", состоящего из одиночного целого числа. Как обычно, вы должны удостовериться, что она обрабатывает числа от 0 до 9 и выдает сообщение об ошибке для чего-либо другого. Это не должно занять у вас много времени! chap4a.frt)

Теперь давайте расширим ее, включив поддержку операций сложения. Измените Expression так:

```
-- ------------------------------------------------------------
-- Parse and translate an expression
: expression ( -- n )
        0 >R
        Look addop? 0= IF  -R getnum >R  ENDIF
        BEGIN   Look addop?
        WHILE   CASE Look
                  '+' OF  '+' match getnum R>      + >R  ENDOF
                  '-' OF  '-' match getnum R> SWAP - >R  ENDOF
                ENDCASE
        REPEAT  R> ;
-- ------------------------------------------------------------
```

Структура Expression, конечно, схожа с тем, что мы делали ранее, так что мы не будем иметь слишком много проблем при ее отладке. Тем не менее это была серьезная разработка, не так ли? Процедуры Add и Subtract исчезли! Причина в том, что для выполнения необходимых действий нужны оба аргумента операции. Я мог бы сохранить эти процедуры и передавать в них значение выражения на данный момент, содержащееся в Value. Но мне показалось более правильным оставить Value как строго локальную переменную, что означает, что код для Add и Subtract должен быть помещен вместе. Этот результат наводит на мысль, что хотя разработанная нами структура была хорошей и проверенной для нашей бесхитростной схемы трансляции, она возможно не могла бы использоваться с ленивой оценкой. Эту небольшую интересную новость нам возможно необходимо иметь в виду в будущем.

Итак, транслятор работает? Тогда давайте сделаем следующий шаг. Несложно понять, что процедура Term должна выглядеть также. Замените каждый вызов GetNum в функции Expression на вызов Term и затем наберите следующую версию Term:

```
-- ------------------------------------------------------------
-- Parse and translate a math term
: term ( -- n )
        getnum >R
        BEGIN   Look mulop?
        WHILE   CASE Look
                  '*' OF '*' match getnum R>      * >R  ENDOF
                  '/' OF '/' match getnum R> SWAP / >R  ENDOF
                ENDCASE
        REPEAT  R> ;
-- ------------------------------------------------------------
```

Теперь испробуйте(chap4c.frt). Не забудьте двух вещей: во-первых мы имеем дело с целочисленным делением, поэтому, например, 1/3 выдаст ноль. Во-вторых, даже если мы можем получать на выходе многозначные числа, входные числа все еще ограничены одиночной цифрой.

Сейчас это выглядит как глупое ограничение, так как мы уже видели как легко может

быть расширена функция GetNum. Так что давайте исправим ее прямо сейчас. Вот новая версия:

```
-- ---------------------------------------------------------------
-- Get a number
: getnum ( -- n )
        Look digit? 0= IF  S" Integer" expected  ENDIF
        0 >R
        BEGIN   Look digit?
        WHILE   Look '0' -  R> #10 * + >R
                getchar
        REPEAT  R> ;
-- ---------------------------------------------------------------
```

Если вы откомпилировали и протестировали эту версию интерпретатора(chap4d.frt), следующим шагом должна быть установка функции Factor, поддерживающей выражения в скобках. Мы задержимся немного дольше на именах переменных. Сначала измените ссылку на GetNum в функции Term, чтобы вместо нее вызывалась функция Factor. Теперь наберите следующую версию Factor:

```
-- ----------------------------------------------------------------
-- Parse and translate a math factor

DEFER expression ( -- )

: factor ( -- n )
        Look '(' = IF  '(' match  expression ')' match  EXIT  ENDIF
        getnum ;
-- ----------------------------------------------------------------
```

Это было довольно легко, а? Мы быстро пришли к полезному интерпретатору(chap4e.frt) .

НЕМНОГО ФИЛОСОФИИ

Прежде чем двинуться дальше, я бы хотел обратить ваше внимание на кое-что. Я говорю о концепции, которую мы использовали на всех этих уроках, но которую я явно не упомянул до сих пор. Я думаю, что пришло время сделать это, так как эта концепция настолько полезная и настолько мощная, что она стирает все различия между тривиально простым синтаксическим анализатором и тем, который слишком сложен для того, чтобы иметь с ним дело.

В ранние дни технологии компиляции люди тратили ужасно много времени на выяснение того, как работать с такими вещами как приоритет операторов┘ способа, который определяет приоритет операторов умножения и деления над сложением и вычитанием и т.п. Я помню одного своего коллегу лет тридцать назад и как возбужденно он выяснял как это делается. Используемый им метод предусматривал создание двух стеков, в которые вы помещали оператор или операнд. С каждым оператором был связан уровень приоритета и правила требовали, чтобы вы фактически выполняли операцию ("уменьшающую" стек) если уровень приоритета на вершине стека был корректным. Чтобы сделать жизнь более интересной оператор типа ")" имел различные приоритеты в зависимости от того, был он уже в стеке или нет. Вы должны были дать ему одно значение перед тем как поместите в стек и другое, когда решите извлечь из стека. Просто для эксперимента я самостоятельно поработал со всем этим несколько лет назад и могу сказать вам, что это очень сложно.

Мы не делали что-либо подобное. Фактически, к настоящему времени синтаксический анализ арифметических выражений должен походить на детскую игру. Как мы оказались настолько удачными? И куда делся стек приоритетов?

Подобная вещь происходит в нашем интерпретаторе выше. Вы просто знаете, что для того, чтобы выполнить вычисления арифметических выражений (в противоположность их анализу), должны иметься числа, помещенные в стек. Но где стек?

Наконец, в учебниках по компиляторам имеются разделы, где обсуждены стеки и другие структуры. В другом передовом методе синтаксического анализа (LR) используется явный стек. Фактически этот метод очень похож на старый способ вычисления арифметических выражений. Другая концепция √ это синтаксическое дерево. Авторы любят рисовать диаграммы из токенов в выражении объединенные в дерево с операторами во внутренних узлах. И снова, где в нашем методе деревья и стеки? Мы не видели ничего такого. Во всех случаях ответ в том, что эти структуры не явные а неявные. В любом машинном языке имеется стек, используемый каждый раз, когда вы вызываете подпрограмму. Каждый раз, когда вызывается подпрограмма, адрес возврата помещается в стек ЦПУ. В конце подпрограммы адрес выталкивается из стека и управление передается на этот адрес. В рекурсивном языке, таком как Pascal, могут также иметься локальные данные, помещенные в стек, и они также возвращаются когда это необходимо.

Например функция Expression содержит локальный параметр, названный Value, которому присваивается значение при вызове Term. Предположим, при следующем вызове Term для второго аргумента, что Term вызывает Factor, который рекурсивно вызывает Expression снова. Этот "экземпляр" Expression получает другое значение для его копии Value. Что случится с первым значением Value? Ответ: он все еще в стеке и будет здесь снова, когда мы возвратимся из нашей последовательности вызовов.

Другими словами, причина, по которой это выглядит так просто в том, что мы максимально использовали ресурсы языка. Уровни иерархии и синтаксические деревья присутствуют здесь, все правильно, но они скрыты внутри структуры синтаксического анализатора и о них заботится порядок в котором вызываются различные процедуры. Теперь, когда вы увидели, как мы делаем это, возможно трудно будет придумать как сделать это каким-либо другим способом. Но я могу сказать вам, что это заняло много лет для создателей компиляторов. Первые компиляторы были слишком сложными. Забавно, как работа становится легче с небольшой практикой.

Вывод из всего того, что я привел здесь, служит и уроком и предупреждением. Урок: дела могут быть простыми если вы приметесь за них с правильной стороны. Предупреждение: смотрите, что делаете. Если вы делаете что-либо самостоятельно и начинаете испытывать потребность в отдельном стеке или дереве, возможно это время спросить себя, правильно ли вы смотрите на вещи. Возможно вы просто не используете возможностей языка так как могли бы.

Следующий шаг √ добавление имен переменных. Сейчас, однако, мы имеем небольшую проблему. В случае с компилятором мы не имели проблем при работе с именами переменных┘ мы просто выдавали эти имена ассемблеру и позволяли остальной части программы заботиться о распределении для них памяти. Здесь же, напротив, у нас должна быть возможность извлекать значения переменных и возвращать их как значение функции Factor. Нам необходим механизм хранения этих переменных.

В ранние дни персональных компьютеров существовал Tiny Basic. Он имел в общей сложности 26 возможных переменных: одна на каждую букву алфавита. Это хорошо соответствует нашей концепции односимвольных токенов, так что мы испробуем этот же прием. В начале интерпретатора, сразу после объявления переменной Look, вставьте строку:

```
CREATE table 26 CELLS ALLOT
```

Мы также должны инициализировать массив, поэтому добавьте следующую процедуру:

```
-- ------------------------------------------------------------
-- Initialize the variable area
: inittable ( -- ) table #26 CELLS ERASE ;
-- ------------------------------------------------------------
```

Вы также должны вставить вызов InitTable в процедуру Init. Не забудьте сделать это, иначе результат может удивить вас!

Теперь, когда у нас есть массив переменных, мы можем модифицировать Factor так, чтобы

он их использовал. Так как мы не имеем (пока) способа для установки значения переменной, Factor будет всегда возвращать для них нулевые значения, но давайте двинемся дальше и расширим его. Вот новая версия:

```
-- --------------------------------------------------------------
-- Parse and translate a math factor

DEFER expression

: factor ( -- n )
        Look '(' = IF  '(' match  expression ')' match  EXIT  ENDIF
        Look alpha? IF  table getname 'A' - CELLS + @
                  ELSE  getnum
                 ENDIF ;
-- --------------------------------------------------------------
```

Как всегда откомпилируйте и протестируйте эту версию программы (chap4f.frt).Даже притом, что все переменные сейчас равны нулю, по крайней мере мы можем правильно анализировать законченные выражения, так же как и отлавливать любые неправильно оформленные.

Я предполагаю вы уже знаете следующий шаг: мы должны добавить операции присваивания, чтобы мы могли помещать что-нибудь в переменные. Сейчас давайте будем "однострочниками", хотя скоро мы сможем обрабатывать множество операторов.

Операция присваивания похожа на то, что мы делали раньше:

```
-- -------------------------------------------------------------
-- Parse and translate an assignment statement
: assignment ( -- )
        getname 'A' - >R
        '=' match
        expression  table R> CELL[] ! ;
-- -------------------------------------------------------------
```

Чтобы протестировать ее, я добавил временный оператор write в основную программу для вывода значения A. Затем я протестировал ее с различными присваиваниями. (chap4g.frt).

Конечно, интерпретируемый язык, который может воспринимать только одну строку программы не имеет большой ценности. Поэтому нам нужно обрабатывать множество утверждений. Это просто означает что необходимо поместить цикл вокруг вызова Assignment. Давайте сделаем это сейчас. Но что должно быть критерием выхода из цикла? Рад, что вы спросили, потому что это поднимает вопрос, который мы были способны игнорировать до сих пор.

Одной из наиболее сложных вещей в любом трансляторе является определение момента когда необходимо выйти из данной конструкции и продолжить выполнение. Пока это не было для нас проблемой, потому что мы допускали только одну конструкцию┘ или выражение или операцию присваивания. Когда мы начинаем добавлять циклы и различные виды операторов, вы найдете, что мы должны быть очень осторожны, чтобы они завершались правильно. Если мы помещаем наш интерпретатор в цикл, то нам нужен способ для выхода из него. В прерывании по концу строки нет ничего хорошего, поскольку с его помощью мы переходим к следующей строке. Мы всегда могли позволить нераспознаваемым символам прерывать выполнение, но это приводило бы к завершению каждой программы сообщением об ошибке, что конечно выглядит несерьезно.

Нам нужен завершающий символ. Я выступаю за завершающую точку в Pascal ("."). Небольшое осложнение состоит в том, что Turbo Pascal завершает каждую нормальную строку двумя символами: возврат каретки (CR) и перевод строки (LF). В конце каждой строки мы должны "съедать" эти символы перед обработкой следующей. Естественным способом было бы сделать это в процедуре Match за исключением того, что сообщение об ошибке Match выводит ожидаемые символы, что для CR и LF не будет выглядеть так хорошо. Для

этого нам нужна специальная процедура, которую мы, без сомнения, будем использовать много раз. Вот она:

```
-- ---------------------------------------------------------------------------
-- Recognize and skip over a newline
: newline ( -- ) Look ^M = IF  getchar Look ^J = IF  getchar  ENDIF  ENDIF ;
-- ---------------------------------------------------------------------------
```

Вставьте эту процедуру в любом удобном месте┘ я поместил ее сразу после Match. Теперь перепишите основную программу, чтобы она выглядела следующим образом:

```
-- ----------------------------------------------------------------
: main ( -- )
        init
        BEGIN
          assignment
          newline
          Look '.' =
        UNTIL ;
-- ----------------------------------------------------------------
```

Обратите внимание, что проверка на CR теперь исчезла и что также нет проверки на ошибку непосредственно внутри NewLine. Это нормально┘ все оставшиеся фиктивные символы будут отловлены в начале следующей операции присваивания.

Хорошо, сейчас мы имеем функционирующий интерпретатор(chap4h.frt). Однако, это не дает нам много хорошего, так как у нас нет никакого способа для ввода или вывода данных. Уверен что нам помогут несколько подпрограмм ввода/вывода!

Тогда давайте завершим этот урок добавив подпрограммы ввода/вывода. Так как мы придерживаемся односимвольных токенов, я буду использовать знак "?" для замены операции чтения, знак "!" для операции записи и символ, немедленно следующий после них, который будет использоваться как односимвольный "список параметров". Вот эти подпрограммы:

```
-- ----------------------------------------------------------------
-- Input routine
: input ( -- )
        '?' match
        table getname 'A' - CELLS + >R
        getnum  R> ! ;
-- ----------------------------------------------------------------
-- Output routine
: output ( -- )
        '!' match
        table getname 'A' - CELLS + @ CR . ;
-- ----------------------------------------------------------------
```

Я полагаю они не очень причудливы┘ например нет никакого символа приглашения при вводе┘ но они делают свою работу(chap4i.frt) .

Соответствующие изменения в основной программе показаны ниже. Обратите внимание, что мы используем обычный прием √ оператор выбора по текущему предсказывающему символу, чтобы решить что делать.

```
-- ----------------------------------------------------------------
-- Main Program
: main ( -- )
        init
        BEGIN
          CASE Look
           '?' OF  input   ENDOF
           '!' OF  output  ENDOF
                   assignment
          ENDCASE
```

```
        newline
        Look '.' =
      UNTIL ;
-- -------------------------------------------------------------
```

Теперь вы закончили создание настоящего, работающего интерпретатора. Он довольно скудный, но работает совсем как "большой мальчик". Он включает три вида операторов (и может различить их!), 26 переменных и операторы ввода/вывода. Единственное, в чем он действительно испытывает недостаток - это операторы управления, подпрограммы и некоторые виды функций для редактирования программы. Функции редактирования программ я собираюсь пропустить. В конце концов, мы здесь не для того, чтобы создать готовый продукт, а чтобы учиться. Управляющие операторы мы раскроем в следующей главе, а подпрограммы вскоре после нее. Я стремлюсь продолжать дальше, поэтому мы оставим интерпретатор в его текущем состоянии.

Я надеюсь, к настоящему времени вы убедились, что ограничение имен одним символом и обработка пробелов это вещи о которых легко позаботиться, как мы сделали это на последнем уроке. На этот раз, если вам захотелось поиграть с этими расширениями, будьте моим гостем┘ они "оставлены как упражнение для студента"(chap4j.frt). Увидимся в следующий раз.

5. Управляющие конструкции

ВВЕДЕНИЕ

В четырех первых главах этой серии мы сконцентрировали свое внимание на синтаксическом анализе математических выражений и операций присваивания. В этой главе мы остановимся на новой и захватывающей теме: синтаксическом анализе и трансляции управляющих конструкций таких как, например, операторы IF.

Эта тема дорога для моего сердца, потому что является для меня поворотной точкой. Я играл с синтаксическим анализом выражений также как мы делали это в этой серии, но я все же чувствовал, что нахожусь еще очень далеко от возможности поддержки полного языка. В конце концов, реальные языки имеют ветвления, циклы, подпрограммы и все такое. Возможно вы разделяли некоторые из этих мыслей. Некоторое время назад, тем не менее, я должен был реализовать управляющие конструкции для структурного препроцессора ассемблера, который я писал. Вообразите мое удивление, когда я обнаружил, что это было гораздо проще, чем синтаксический анализ выражений, через который я уже прошел. Я помню подумал "Эй, это же просто!". После того, как мы закончим этот урок, я готов поспорить, что вы будете думать так же.

ПЛАН

Далее мы снова начнем с пустого Cradle и, как мы делали уже дважды до этого, будем строить программу последовательно. Мы также сохраним концепцию односимвольных токенов, которая так хорошо служила нам до настоящего времени. Это означает, что "код" будет выглядеть немного забавным с "i" вместо IF, "w" вместо WHILE и т.д. Но это поможет нам узнать основные понятия не беспокоясь о лексическом анализе. Не бойтесь... в конечном счете мы увидим что-то похожее на "настоящий" код.

Я также не хочу, чтобы мы увязли в работе с какими либо операторами кроме ветвлений, такими как операции присваивания, с которыми мы уже работали. Мы уже показали, что можем обрабатывать их, так что нет никакого смысла таскать этот лишний багаж в течение предстоящих занятий. Вместо этого я буду использовать анонимный оператор "other" для замены неуправляющих операторов. Мы должны генерировать для них некоторый обьектный код (мы возвращаемся к компиляции а не интерпретации), так что за неимением чего-либо

другого я буду просто повторять входной символ.

Итак, тогда, начав с еще одной копии Cradle, давайте определим процедуру:

```
-- ---------------------------------------------------------------
-- Recognize and translate an "other"
: other ( -- ) getname LOCAL s  'OF s 1 emitln ;
-- ---------------------------------------------------------------
```

Теперь включим ее вызов в основную программу таким образом:

```
-- ---------------------------------------------------------------
: main ( -- ) init other ;
-- ---------------------------------------------------------------
```

Запустите программу и посмотрите, что вы получили(chap5a.frt). Не очень захватывающе, не так ли? Но не зацикливайтесь на этом, это только начало, результат будет лучше.

Первое, что нам нужно - это возможность работать с более чем одним оператором, так как однострочные ветвления довольно ограничены. Мы делали это на последнем занятии по интерпретации, но сейчас давайте будем немного более формальными. Рассмотрите следующую БНФ:

<program> ::= <block> END

<block> ::= [<statement>]*

Это означает, что программа определена как блок, завершаемый утверждением END. Блок, в свою очередь, состоит из нуля или более операторов. Пока у нас есть только один вид операторов.

Что является признаком окончания блока? Это просто любая конструкция, не являющаяся оператором "other". Сейчас это только утверждение END.

Вооружившись этими идеями, мы можем приступать к созданию нашего синтаксического анализатора. Код для program (мы должны назвать его DoProgram, иначе Pascal будет ругаться) следующий:

```
-- ---------------------------------------------------------------
-- Parse and translate a program
: doprogram ( -- )
   block
   Look 'e' <> IF  S" End" expected  ENDIF
   S" END" emitln ;
-- ---------------------------------------------------------------
```

Обратите внимание, что я выдаю ассемблеру команду "END", что своего рода расставляет знаки препинания в выходном коде и заставляет чувствовать, что мы анализируем здесь законченную программу.

Код для Block:

```
-- ---------------------------------------------------------------
-- Recognize and translate a statement block
: block ( -- )
   BEGIN  Look 'e' <>
   WHILE  other
   REPEAT ;
-- ---------------------------------------------------------------
```

(Из формы процедуры вы видите, что мы собираемся постепенно ее расширять!)

OK, вставьте эти подпрограммы в вашу программу. Замените вызов Block в основной

программе на вызов DoProgram. Теперь испытайте ее и посмотрите как она работает.(chap5b.frt) Хорошо, все еще не так много, но мы становимся все ближе.

НЕМНОГО ОСНОВ

Прежде чем мы начнем определять различные управляющие конструкции, мы должны положить немного более прочное основание. Во-первых, предупреждаю: я не буду использовать для этих конструкций тот же самый синтаксис с которым вы знакомы по Паскалю или Си. К примеру синтаксис Паскаль для IF такой:

IF <condition> THEN <statement>

(где <statement>, конечно, может быть составным.)

Синтаксис C аналогичен этому:

IF (<condition>) <statement>

Вместо этого я буду использовать нечто более похожее на Ada:

IF <condition> <block> ENDIF

Другими словами, конструкция IF имеет специфический символ завершения. Это позволит избежать висячих else Паскаля и Си и также предотвращает необходимость использовать скобки {} или begin-end. Синтаксис, который я вам здесь показываю, фактически является синтаксисом языка KISS, который я буду детализировать в следующих главах. Другие конструкции также будут немного отличаться. Это не должно быть для вас большой проблемой. Как только вы увидите, как это делается, вы поймете, что в действительности не имеет большого значения, какой конкретный синтаксис используется. Как только синтаксис определен, включить его в код достаточно просто.

Теперь, все конструкции, с которыми мы будем иметь дело, включают передачу управления, что на уровне ассемблера означает условные и/или безусловные переходы. К примеру простой оператор IF:

IF <condition> A ENDIF B...

должен быть переведен в:

```
    Если условие не выполнено то переход на L
    A
L:  B
    ...
```

Ясно, что нам понадобятся несколько процедур, которые помогут нам работать с этими переходами. Ниже я определил две из них. Процедура NewLabel генерирует уникальные метки. Это сделано с помощью простого способа называть каждую метку 'Lnn', где nn - это номер метки, начинающийся с нуля. Процедура PostLabel просто выводит метки в соответствующем месте.

Вот эти две подпрограммы:

```
-- -------------------------------------------------------------
-- Generate a unique label
: newlabel ( -- c-addr u )
        S" @@" Lcount U>D <# #S #> $+
        1 +TO Lcount ;
-- -------------------------------------------------------------
-- Post a label to output
: postlabel ( c-addr u -- )
        CR TYPE ':' EMIT ;
-- -------------------------------------------------------------
```

Заметьте, что мы добавили новую глобальную переменную LCount, так что вы должны изменить раздел описания переменных в начале программы, следующим образом:

```
0 VALUE Lcount  -- label counter
```

Также добавьте следующий дополнительный инициализирующий код в Init:

```
: init ( -- ) init CLEAR Lcount ;
```

(Не забудьте сделать это, иначе ваши метки будут выглядеть действительно странными!).

В этом месте я также хотел бы показать вам новый вид нотации. Если вы сравните форму оператора IF, указанную выше, с ассемблерным кодом, который должен быть получен, то вы можете увидеть, что существуют некоторые определенные действия, связанные с каждым ключевым словом в операторе:

IF: Сначала получить условие и выдать код для него. Затем создать уникальную метку и выдать переход если условие ложно.

ENDIF: Выдать метку.

Эти действия могут быть показаны очень кратко, если мы запишем синтаксис таким образом:

```
IF
<condition> { Condition;
              L = NewLabel;
              Emit(Branch False to L); }
<block>
ENDIF       { PostLabel(L) }
```

Это пример синтаксически-управляемого перевода. Мы уже делали все это... мы просто никогда прежде не записывали это таким образом. Содержимое фигурных скобок представляет собой действия, которые будут выполняться. Хорошо в этом способе представления то, что он не только показывает что мы должны распознать, но также и действия, которые мы должны выполнить и в каком порядке. Как только мы получаем такой синтаксис, код возникает почти сам собой.

Почти единственное, что осталось сделать - конкретизировать то, что мы подразумеваем под "Переход если условие ложно".

Я полагаю, что должен быть код, выполняющийся для <condition>, который будет выполнять булеву алгебру и вычислять некоторый результат. Он также должен установить флажки условий, соответствующие этому результату. Теперь, обычным соглашением для булевых переменных является использование 0000 для представления значения "ложь" и какого-либо другого значения (кто-то использует FFFF, кто-то 0001) для представления "истины".

Процессор 68000 устанавливает флажки условий всякий раз, когда любые данные перемещаются или рассчитываются. Если данные равны 0000 (что соответствует условию ложь, запомните) будет установлен флажок ноль. Код для "перехода по нулю" - BEQ. Таким образом

```
  JE  <=> Branch if false
  JNE <=> Branch if true
```

По природе вещей большинство ветвлений, которые мы увидим, будут JE... мы будем обходить вокруг кода, который должен выполняться когда условие истинно.

ОПЕРАТОР IF

После этого небольшого пояснения метода мы наконец готовы начать программирование синтаксического анализатора для условного оператора. Фактически, мы уже почти сделали это! Как обычно я буду использовать наш односимвольный подход, с символом "i" вместо "IF" и "e" вместо "ENDIF" (также как и END... это двойственная природа не вызывает никакого беспорядка). Я также пока полностью пропущу символ для условия ветвления, который мы все еще должны определить.

Код для DoIf:

```
-- -------------------------------------------------------------
-- Recognize and translate an IF construct

DEFER block

: doIF ( -- )
        'i' match
        newlabel $TEMP L
        condition
        L S"  offset NEAR  je," $+ emitln
        block
        'e' match
        L postlabel
        L $FREE ;
-- -------------------------------------------------------------
```

Добавьте эту подпрограмму в вашу программу и измените Block так, чтобы он ссылался на нее как показано ниже:

```
-- -------------------------------------------------------------
-- Recognize and translate a statement block
:NONAME ( -- )
   BEGIN Look 'e' <>
   WHILE CASE Look
           'i' OF doIF  ENDOF
                  other
         ENDCASE
   REPEAT ; IS block
-- -------------------------------------------------------------
```

Обратите внимание на обращение к процедуре Condition. В конечном итоге мы напишем подпрограмму, которая сможет анализировать и транслировать любое логическое условие которое мы ей дадим. Но это уже тема для отдельной главы (фактически следующей). А сейчас давайте просто заменим ее макетом, который выдает некоторый текст. Напишите следующую подпрограмму:

```
-- -------------------------------------------------------------
-- Parse and translate a boolean condition
-- This version is a dummy
: condition ( -- )  S" <condition>" emitln ;
-- -------------------------------------------------------------
```

Вставьте эту процедуру в вашу программу как раз перед DoIf. Теперь запустите программу. Испробуйте строку типа:

aibece

Как вы можете видеть, синтаксический анализатор, кажется, распознает конструкцию и вставляет объектный код в правильных местах. Теперь попробуйте набор вложенных IF:

aibicedefe

Он начинает все более походить на настоящий, не так ли?

Теперь, когда у нас есть общая идея (и инструменты такие как нотация и процедуры NewLabel и PostLabel) проще пареной репы расширить синтаксический анализатор для поддержки и других конструкций. Первое (а также и одно из самых сложных) это добавление условия ELSE в IF. В БНФ это выглядит так:

IF <condition> <block> [ELSE <block>] ENDIF

Сложность возникает просто потому, что здесь присутствует необязательное условие, которого нет в других конструкциях.

Соответствующий выходной код должен быть таким:

```
     <condition>
     L1 offset NEAR je,
     <block>
     L2 offset NEAR jmp,
L1:  <block>
L2:  ...
```

Это приводит нас к следующей синтаксически управляемой схеме перевода:

```
IF
<condition>    { L1 = NewLabel
                 L2 = NewLabel
                 Emit(L1 offset NEAR je,) }
<block>
ELSE           { Emit(L2 offset NEAR jmp,)
                 PostLabel(L1) }
<block>
ENDIF          { PostLabel(L2) }
```

Сравнение этого со случаем IF без ELSE дает нам понимание того, как обрабатывать обе эти ситуации. Код ниже выполняет это. (Обратите внимание, что использую "l" вместо "ELSE" так как "e" имеет другое назначение):

```
-- ---------------------------------------------------------------
-- Recognize and translate an IF construct
: doIF ( -- )
        0 0 LOCALS| L2 L1 |
        'i' match
        condition
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L1 PACK DROP
        L1 COUNT DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP
        L1 COUNT S"  offset NEAR  je," $+ emitln
        block
        Look 'l' = IF   'l' match
                        L2 FREE ?ALLOCATE
                        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP
                        L2 COUNT S"  offset NEAR  jmp," $+ emitln
                        L1 COUNT postlabel
                        block
                ENDIF
        'e' match
        L2 COUNT postlabel

        L1 FREE ?ALLOCATE
        L2 FREE ?ALLOCATE ;
-- ---------------------------------------------------------------
```

Вы получили его. Законченый анализатор/транслятор в 19 строк кода.

Сейчас протестируйте его. (chap5d.frt).и спробуйте что-нибудь типа:

aiblcede

Работает? Теперь, только для того, чтобы убедиться, что мы ничего не испортили и случай с IF без ELSE тоже будет обрабатываться, введите

aibece

Теперь испробуйте несколько вложенных IF. Испытайте что-нибудь на ваш выбор, включая несколько неправильных утверждений. Только запомните, что 'e' не является допустимым оператором "other".

ОПЕРАТОР WHILE

Следующий вид оператора должен быть простым, так как мы уже имеем опыт. Синтаксис, который я выбрал для оператора WHILE следующий:

WHILE <condition> <block> ENDWHILE

Знаю, знаю, мы действительно не нуждаемся в отдельных видах ограничителей для каждой конструкции... вы можете видеть, что фактически в нашей односимвольной версии 'e' используется для всех из них. Но я также помню множество сессий отладки в Паскале, пытаясь отследить своенравный END который по мнению компилятора я хотел поместить где-нибудь еще. По своему опыту знаю, что специфичные и уникальные ключевые слова, хотя они и добавляются к словарю языка, дают небольшую защиту от ошибок, которая стоит дополнительной работы создателей компиляторов.

Теперь рассмотрите, во что должен траслироваться WHILE:

```
L1:  <condition>
     L2 offset NEAR je,
     <block>
     L1 offset NEAR jmp,
L2:
```

Как и прежде, сравнение этих двух представлений дает нам действия, необходимые на каждом этапе:

```
WHILE          { L1 = NewLabel
                 PostLabel(L1) }
<condition>    { Emit(L2 offset NEAR je,) }
<block>
ENDWHILE       { Emit(L1 offset NEAR jmp,)
                 PostLabel(L2) }
```

Код выходит непосредственно из синтаксиса:

```
-- ---------------------------------------------------------------
-- Parse and translate a while statement
: doWHILE ( -- )
        0 0 LOCALS| L2 L1 |
        'w' match
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L1 PACK DROP
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP

        L1 COUNT postlabel
        condition
        L2 COUNT S"  offset NEAR  je,"  $+ emitln
        block
        'e' match
        L1 COUNT S"  offset NEAR  jmp," $+ emitln
        L2 COUNT postlabel

        L1 FREE ?ALLOCATE
        L2 FREE ?ALLOCATE ;
```

```
-- ------------------------------------------------------------
```

Так как мы получили новый оператор, мы должны добавить его вызов в процедуру Block:

```
-- ------------------------------------------------------------
-- Recognize and translate a statement block
: block ( -- )
        BEGIN   Look 'e' <>
                Look 'l' <> AND
        WHILE   CASE Look
                 'i' OF  doIF      ENDOF
                 'w' OF  doWHILE   ENDOF
                         other
               ENDCASE
        REPEAT ;
-- ------------------------------------------------------------
```

Никаких других изменений не требуется.

Хорошо, протестируйте новую программу.(chap5e.frt). Заметьте, что на этот раз код <condition> находится внутри верхней метки, как раз там, где нам надо. Попробуйте несколько вложенных циклов. Испробуйте циклы внутри IF и IF внутри циклов. Если вы немного напутаете то, что вы должны набирать, не смущайтесь: вы пишите ошибки и в других языках, не правда ли? Код будет выглядеть более осмысленным, когда мы получим полные ключевые слова.

Я надеюсь, что к настоящему времени вы начинаете понимать, что это действительно просто. Все, что нам необходимо было сделать для того, чтобы создать новую конструкцию, это разработать ее синтаксически-управляемый перевод. Код возникает из него, и это не влияет на другие подпрограммы. Как только вы почувствуете это, вы увидите, что можете добавлять новые конструкции почти также быстро, как вы можете их придумывать.

ОПЕРАТОР LOOP

Мы могли бы остановиться на этом и иметь работающий язык. Много раз было показано, что языка высокого уровня всего с двумя конструкциями IF и WHILE достаточно для написания структурного кода. Но раз уж мы начали, то давайте немного расширим репертуар.

Эта конструкция даже проще, так как она совсем не имеет проверки условия... это бесконечный цикл. Имеет ли смысл такой цикл? Немного сам по себе, но позднее мы собираемся добавить команду BREAK, которая даст нам способ выхода из цикла. Она делает язык значительно более богатым, чем Паскаль, который не имеет команды выхода из цикла и также позволяет избежать забавных конструкций типа WHILE(1) или WHILE TRUE в C и Паскале.

Синтаксис прост:

LOOP <block> ENDLOOP

Синтаксически управляемый перевод:

```
LOOP          { L = NewLabel
                PostLabel(L) }
<block>
ENDLOOP       { Emit(L offset NEAR jmp, }
```

Соответствующий код показан ниже. Так как мы уже использовали "l" для ELSE на этот раз я использовал последнюю букву "p" как "ключевое слово".

```
-- ------------------------------------------------------------
-- Parse and translate a loop statement
: doLOOP ( -- )
```

```
        0 LOCAL L
        'p' match
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L PACK DROP
        L COUNT postlabel
        block
        'e' match
        L COUNT S"  offset NEAR  jmp," $+ emitln
        L FREE ?ALLOCATE ;
-- ---------------------------------------------------------------
```

После того, как вы вставите эту подпрограмму, не забудьте добавить строчку в Block для ее вызова(chap5f.frt) .

REPEAT-UNTIL

Имеется одна конструкция, которую я взял напрямую из Паскаля. Синтаксис:

REPEAT <block> UNTIL <condition>

и синтаксически-управляемый перевод:

REPEAT { L = NewLabel;

PostLabel(L) }

<block>

UNTIL

<condition> { Emit(BEQ L) }

Как обычно, код вытекает отсюда довольно легко:

```
-- ---------------------------------------------------------------
-- Parse and translate a REPEAT statement
: doREPEAT ( -- )
        0 LOCAL L
        'r' match
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L PACK DROP
        L COUNT postlabel
        block
        'u' match
        condition
        L COUNT S"  offset NEAR  je," $+ emitln
        L FREE ?ALLOCATE ;
-- ---------------------------------------------------------------
```

Как и прежде, мы должны добавить вызов DoRepeat в Block. Хотя на этот раз есть различия. Я решил использовать "r" вместо REPEAT (естественно), но я также решил использовать "u" вместо UNTIL. Это означает, что "u" должен быть добавлен к множеству символов в условии while. Это символы, которые сигнализируют о выходе из текущего блока... символы "follow", на жаргоне разработчиков компиляторов.

```
-- ---------------------------------------------------------------
-- Recognize and translate a statement block
: block ( -- )
        BEGIN   Look 'e' <>
                Look 'l' <> AND
                Look 'u' <> AND
        WHILE   CASE Look
                 'i' OF  L doIF    ENDOF
                 'w' OF  doWHILE   ENDOF
                 'p' OF  doLOOP    ENDOF
                 'r' OF  doREPEAT  ENDOF
                         other
               ENDCASE
        REPEAT ;
```

```
-- ------------------------------------------------------------
```

ЦИКЛ FOR

Цикл FOR очень удобен, но он тяжел для трансляции. Не столько потому, что сама конструкция трудна... в конце концов это всего лишь цикл... но просто потому, что она трудна для реализации на ассемблере. Как только код придуман, трансляция достаточно проста.

Фаны Си любят цикл FOR этого языка (фактически он проще для кодирования), но вместо него я выбрал синтаксис очень похожий на синтаксис из старого доброго Бейсика:

```
FOR <ident> = <expr1> TO <expr2> <block> ENDFOR
```

Сложность трансляции цикла "FOR" зависит от выбранного вами способа его реализации, от пути, которым вы решили определять правила обработки ограничений. Рассчитывается ли expr2 каждый раз при прохождении цикла, например, или оно обрабатывается как постоянное ограничение? Всегда ли вы проходите цикл хотя бы раз, как в Fortran, или нет. Все становится проще, если вы приверженец точки зрения что эта конструкция эквивалентна:

```
<ident> = <expr1>
TEMP = <expr2>
WHILE <ident> <= TEMP
<block>
ENDWHILE
```

Заметьте, что с этим определением цикла <block> не будет выполнен вообще если <expr1> изначально больше чем <expr2>.

Код 68000, необходимый для этого, сложней чем все что мы делали до сих пор. Я сделал несколько попыток, помещая и счетчик и верхний предел в стек, в регистры и т.д. В конечном итоге я остановился на гибридном варианте размещения, при котором счетчик помещается в памяти (поэтому он может быть доступен внутри цикла) а верхний предел - в стеке. Оттранслированный код получился следующий:

```
     <ident>                         get name of loop counter
     <expr1>                         get initial value
     1 b# -> eax  sub,               predecrement it
     eax -> ident dword-ptr  mov,    save it
     <expr2>                         get upper limit
     eax push,                       save it on stack

L1:  ident dword-ptr -> eax  mov,    address loop counter to EAX
     1 b# -> eax add,                bump the counter
     eax -> ident dword-ptr  mov,    save new value
     [esp] -> eax cmp,               check for range
     L2 offset NEAR jg,              skip out if D0 > (SP)
     <block>
     L1 offset NEAR jmp,             loop for next pass
L2:  [esp 4 +] -> esp  lea,          clean up the stack
```

Ничего себе! Это же куча кода... строка, содержащая <block> кажется совсем потерявшейся. Но это лучшее из того, что я смог придумать. Я полагаю, чтобы вам помочь, вы должны иметь в виду что в действительности это всего лишь шестнадцать слов, в конце концов. Если кто-нибудь сможет оптимизировать это лучше, пожалуйста дайте мне знать.

Однако, подпрограмма анализа довольно проста теперь, когда у нас есть код:

```
-- ------------------------------------------------------------
-- Parse and translate a for statement
: doFOR ( -- )
        0 0 0 LOCALS| name L2 L1 |
        'f' match
```

```
        getname TO name
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L1   PACK DROP
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2   PACK DROP
        '=' match
        expression
        S" 1 b# -> eax sub," emitln
        S" eax -> " name CHAR-APPEND S"  dword-ptr mov," $+ emitln
        expression
        S" eax push," emitln
        L1 COUNT postlabel
        S"  dword-ptr -> eax mov," name CHAR-PREPEND emitln
        S" 1 b# -> eax add, eax -> " name CHAR-APPEND  S"  dword-ptr mov," $+ emitln
        S" [esp] -> eax cmp," emitln
        L2 COUNT S"  offset NEAR  jg,"  $+ emitln
        block
        'e' match
        L1 COUNT S"  offset NEAR  jmp," $+ emitln
        L2 COUNT postlabel
        S" [esp 4 +] -> esp lea," emitln
        L1   FREE ?ALLOCATE
        L2   FREE ?ALLOCATE ;
-- --------------------------------------------------------------
```

Так как в этой версии синтаксического анализатора у нас нет выражений, я использовал тот же самый прием что и для Condition и написал подпрограмму:

```
-- --------------------------------------------------------------
-- Parse and translate an expression
-- This version is a dummy
: expression ( -- ) S" <expr>" emitln ;
-- --------------------------------------------------------------
```

Испытайте его. Снова, не забудьте добавить вызов в Block. Так как у нас нет возможности ввода для фиктивной версии Expression, типичная входная строка будет выглядеть так:

afi=bece

Хорошо, генерируется много кода, не так ли? Но, по крайней мере, это правильный код.

ОПЕРАТОР DO

Из-за всего этого мне захотелось иметь более простую версию цикла FOR. Причина появления всего этого кода выше состоит в необходимости иметь счетчик цикла, доступный как переменная внутри цикла. Если все, что нам нужно это считающий цикл, позволяющий нам выполнить что-то определенное число раз, но не нужен непосредственный доступ к счетчику, имеется более простое решение. Процессор 68000 имеет встроенную команду "уменьшить и переход если не ноль", которая является идеальной для подсчета. Для полноты давайте добавим и эту конструкцию. Это будет наш последний цикл.

Синтаксис и его перевод:

```
  DO
  <expr>         { L = NewLabel
                   PostLabel(L)
                   Emit( eax push, ) }
  <block>
  ENDDO          { Emit( eax pop,  )
                   Emit( 1 b# -> eax sub, L offset NEAR  jnz, ) }
```

Это гораздо проще! Цикл будет выполняться <expr> раз. Вот код:

```
-- --------------------------------------------------------------
```

```
-- Parse and Translate a DO Statement
: doDO ( -- )
        0 LOCAL L
        'd' match
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L PACK DROP
        expression
        L COUNT postlabel
        S" eax push,"             emitln
        block
        'e' match
        S" eax pop,"              emitln
        S" 1 b# -> eax sub,"      emitln
        L COUNT
        S"  offset NEAR  jnz," $+ emitln
        L FREE ?ALLOCATE  ;
-- --------------------------------------------------------------
```

Я думаю вы согласитесь, что это гораздо проще, чем классический цикл FOR. Однако, каждая конструкция имеет свое назначение.

ОПЕРАТОР BREAK

Ранее я обещал вам оператор BREAK для сопровождения цикла LOOP. Им я в некотором роде горд. На первый взгляд BREAK кажется действительно сложным. Моим первым подходом было просто использовать его как дополнительный ограничитель в Block и разделить все циклы на две части точно также как я сделал это для ELSE оператора IF. Но, оказывается, это не работает, потому что оператор BREAK редко находится на том же самом уровне, что и сам цикл. Наиболее вероятное место для BREAK - сразу после IF, что приводило бы к выходу из конструкции IF, а не из окружающего цикла. Неправильно. BREAK должен выходить из внутреннего LOOP даже если он вложен в несколько уровней IF.

Моей следующей мыслью было просто сохранять в какой-то глобальной переменной, метку окончания самого вложенного цикла. Это также не работает, потому что может возникнуть прерывание из внутреннего цикла с последующим прерыванием из внешнего. Сохранение метки для внутреннего цикла затерло бы метку для внешного. Так глобальная переменная превратилась в стек. Дело становилось грязным.

Тогда я решил последовать своему собственному совету. Помните последний урок, когда я показал вам как хорошо служит нам неявный стек синтаксического анализатора с рекурсивным спуском. Я сказал, что если вы начинаете видеть потребность во внешнем стеке, возможно вы делаете что-то неправильно. Действительно возможно заставить рекурсию, встроенную в наш синтаксический анализатор, позаботиться обо всем и это решение настолько простое, что кажется удивительным.

Секрет состоит в том, чтобы заметить, что каждый оператор BREAK должен выполняться внутри блока... и ни в каком другом месте. Так что все, что мы должны сделать это передать в Block адрес выхода из самого внутреннего цикла. Затем он может передать этот адрес подпрограмме, транслирующей инструкцию Break. Так как оператор IF не изменяет уровень цикла, процедура DoIf не должна делать что-либо за исключением передачи метки в ее блок (оба из них). Так как циклы изменяют уровень, каждый цикл просто игнорирует любую метку выше его и передает свою собственную метку выхода дальше.

Все это проще показать вам чем описывать. Я продемонстрирую это с самым простым циклом, циклом LOOP:

```
-- --------------------------------------------------------------
-- Parse and Translate a LOOP Statement
: doLOOP ( -- )
        0 0 LOCALS| L2 L1 |
        'p' match
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L1 PACK DROP
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP
```

```
        L1 COUNT postlabel
        L2 block
        'e' match
        L1 COUNT S"  offset NEAR  jmp," $+ emitln
        L2 COUNT postlabel

        L1 FREE ?ALLOCATE
        L2 FREE ?ALLOCATE ;
-- ---------------------------------------------------------------
```

Заметьте, что теперь DoLoop имеет две метки а не одну. Вторая дает команде BREAK адрес перехода Если в цикле нет BREAK, то мы зря потратили метку и немного загромоздили код, но не нанесли никакого вреда.

Заметьте также, что процедура Block теперь имеет параметр, который для циклов всегда будет адресом выхода. Новая версия Block:

```
-- ---------------------------------------------------------------
-- Recognize and Translate a Statement Block
: block ( label -- )
        LOCAL L
        BEGIN   Look 'e' <>
                Look 'l' <> AND
                Look 'u' <> AND
        WHILE   CASE Look
                'i' OF  L doIF    ENDOF
                'w' OF  doWHILE   ENDOF
                'p' OF  doLOOP    ENDOF
                'r' OF  doREPEAT  ENDOF
                'f' OF  doFOR     ENDOF
                'd' OF  doDO      ENDOF
                'b' OF  L doBREAK ENDOF
                        other
               ENDCASE
        REPEAT ;
-- ---------------------------------------------------------------
```

Снова заметьте, что все что Block делает с меткой это передает ее в DoIf и DoBreak. Циклы не нуждаются в ней, потому что они в любом случае передают свою собственную метку.

Новая версия DoIf:

```
-- ---------------------------------------------------------------
-- Recognize and translate an IF construct

DEFER block ( L -- )

: doIF ( label -- )
        0 0 LOCALS| L2 L1 L |
        'i' match
        condition
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L1 PACK DROP
        L1 COUNT DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP
        L1 COUNT S"  offset NEAR  je," $+ emitln
        L block
        Look 'l' = IF   'l' match
                        L2 FREE ?ALLOCATE
                        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP
                        L2 COUNT S"  offset NEAR  jmp," $+ emitln
                        L1 COUNT postlabel
                        L block
                ENDIF
        'e' match
        L2 COUNT postlabel
```

```
        L1 FREE ?ALLOCATE
        L2 FREE ?ALLOCATE ;
-- ---------------------------------------------------------------
```

Здесь единственное, что изменяется, это добавляется параметр у процедуры Block. Оператор IF не меняет уровень вложенности цикла, поэтому DoIf просто передает метку дальше. Независимо от того, сколько уровней вложенности IF мы имеем, будет использоваться та же самая метка.

Теперь не забудьте, что DoProgram также вызывает Block и теперь необходимо передавать ей метку. Попытка выхода из внешнего блока является ошибкой, поэтому DoProgram передает пустую метку, которая перехватывается DoBreak:

```
-- ---------------------------------------------------------------
-- Recognize and translate a BREAK
: doBREAK ( label -- )
        'b' match
        DUP C@ 0= IF  S" No loop to break from" aborts  ENDIF
        COUNT S"  offset NEAR  jmp," $+ emitln ;
-- ---------------------------------------------------------------
-- Parse and translate a program
: doprogram ( -- )
        C" " block
        Look 'e' <> IF  S" End" expected EXIT  ENDIF
        S" END" emitln ;
-- ---------------------------------------------------------------
```

Этот код позаботится почти обо всем. Испытайте его, посмотрите, сможете ли вы "сломать" ("break") его (каламбур). Аккуратней однако. К настоящему времени мы использовали так много букв, что трудно придумать символ, который не представляет сейчас какое либо зарезервированное слово. Не забудьте, перед тем, как вы протестируете программу, вы должны будете исправить каждый случай появления Block в других циклах для включения нового параметра. Сделайте это точно так же, как я сделал это для LOOP.

Я сказал выше "почти". Есть одна небольшая проблема: если вы внимательно посмотрите на код, генерируемый для DO, вы увидите, что если вы прервете этот цикл, то значение счетчика все еще остается в стеке. Мы должны исправить это! Позор... это была одна из самых маленьких наших подпрограмм, но это не помогло. Вот новая версия, которая не имеет этой проблемы:

```
-- ---------------------------------------------------------------
-- Parse and translate a DO statement
: doDO ( -- )
        0 0 LOCALS| L2 L1 |
        'd' match
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L1 PACK DROP
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP
        expression
        L1 COUNT postlabel
        S" eax push,"        emitln
        L2 block
        'e' match
        S" eax pop,"         emitln
        S" 1 b# -> eax sub," emitln
        L1 COUNT S"  offset NEAR  jnz," $+ emitln
        S" [esp -4 +] -> esp lea," emitln
        L2 COUNT postlabel
        S" [esp  4 +] -> esp lea," emitln
        L1 FREE ?ALLOCATE
        L2 FREE ?ALLOCATE ;
-- ---------------------------------------------------------------
```

Две дополнительные инструкции LEA -4 и LEA +4 заботятся о сохранении стека в правильной форме.

ЗАКЛЮЧЕНИЕ

К этому моменту мы создали ряд управляющих конструкций… в действительности более богатый набор чем предоставляет почти любой другой язык программирования. И, за исключением цикла FOR, это было довольно легко сделать. Но даже этот цикл был сложен только потому, что сложность заключалась в ассемблере.

Я завершаю на этом урок. Чтобы можно было обернуть наш продукт красной ленточкой, в действительности мы должны иметь настоящие ключевые слова вместо этих игрушечных односимвольных. Вы уже видели, что расширить компилятор для поддержки многосимвольных слов не трудно, но в этом случае возникнут большие различия в представлении нашего входного кода. Я оставлю этот небольшой кусочек для следующей главы. В этой главе мы также рассмотрим логические выражения, так что мы сможем избавиться от фиктивной версии Condition, которую мы здесь использовали. Увидимся.

Для справочных целей привожу полный текст синтаксического анализатора для этого урока:

```
-- Variable declarations -------------------------------------------------------

0 VALUE Look             -- lookahead character
0 VALUE Lcount           -- label counter

-- Tools -----------------------------------------------------------------------

: getchar  ( -- ) EKEY TO Look ;           -- read new character from input stream
: error    ( c-addr u -- ) CR ^G EMIT ." Error: " TYPE ." ." ;  -- report an error
: aborts   ( c-addr u -- ) error ABORT ;                -- report error and halt
: expected ( c-addr u -- ) S"  Expected" $+ aborts ;  -- report what was expected
: alpha?   ( char -- tf )  >UPC 'A' 'Z' 1+ WITHIN ; -- recognize an alpha character
: digit?   ( char -- tf )  '0' '9' 1+ WITHIN ;      -- recognize a decimal digit
: emits    ( c-addr u -- ) Tab EMIT TYPE ;          -- output a string with tab
: emitln   ( c-addr u -- ) CR emits ;         -- output a string with tab and crlf
: init     ( -- )          CLEAR Lcount  CR getchar ;   -- initialize

-- Specifics -------------------------------------------------------------------

-- match a specific input character
: match ( char -- )
        DUP Look = IF  DROP getchar
                 ELSE  S" `" ROT CHAR-APPEND  &' CHAR-APPEND expected
                ENDIF ;

-- get an identifier
: getname ( -- c-addr u )
        Look alpha? 0= IF  S" Name" expected  ENDIF
        Look >UPC  S" " ROT CHAR-APPEND
        getchar ;

-- get a number
: getnum ( -- char )
        Look digit? 0= IF  S" Integer" expected  ENDIF
        Look
        getchar ;

: other ( -- ) getname emitln ;

-- generate a unique label
: newlabel ( -- c-addr u ) S" @@" Lcount U>D <# #S #> $+  1 +TO Lcount ;

-- post a label to output
: postlabel ( c-addr u -- ) CR TYPE ':' EMIT ;
```

```
DEFER block

-- parse and translate a boolean condition
: condition  ( -- ) S" " emitln ;

-- parse and translate a math expression
: expression ( -- ) S" " emitln ;

-- recognize and translate an IF construct
: doIF ( label -- )
        0 0 LOCALS| L2 L1 L |
        'i' match
        condition
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L1 PACK DROP
        L1 COUNT DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP
        L1 COUNT S"  offset NEAR  je," $+ emitln
        L block
        Look 'l' = IF   'l' match
                        L2 FREE ?ALLOCATE
                        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP
                        L2 COUNT S"  offset NEAR  jmp," $+ emitln
                        L1 COUNT postlabel
                        L block
                ENDIF
        'e' match
        L2 COUNT postlabel
        L1 FREE ?ALLOCATE
        L2 FREE ?ALLOCATE ;

-- recognize and translate a WHILE construct
: doWHILE ( -- )
        0 0 LOCALS| L2 L1 |
        'w' match
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L1 PACK DROP
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP
        L1 COUNT postlabel
        condition
        L2 COUNT S"  offset NEAR  je,"  $+ emitln
        L2 block
        'e' match
        L1 COUNT S"  offset NEAR  jmp," $+ emitln
        L2 COUNT postlabel
        L1 FREE ?ALLOCATE
        L2 FREE ?ALLOCATE ;

-- recognize and translate a LOOP construct
: doLOOP ( -- )
        0 0 LOCALS| L2 L1 |
        'p' match
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L1 PACK DROP
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP
        L1 COUNT postlabel
        L2 block
        'e' match
        L1 COUNT S"  offset NEAR  jmp," $+ emitln
        L2 COUNT postlabel
        L1 FREE ?ALLOCATE
        L2 FREE ?ALLOCATE ;

-- recognize and translate a FOR construct
: doFOR ( -- )
        0 0 0 LOCALS| name L2 L1 |
        'f' match
        getname TO name
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L1   PACK DROP
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2   PACK DROP
```

```
        '=' match
        expression
        S" 1 b# -> eax sub," emitln
        S" eax -> " name CHAR-APPEND S"  dword-ptr mov," $+ emitln
        expression
        S" eax push," emitln
        L1 COUNT postlabel
        S"  dword-ptr -> eax mov," name CHAR-PREPEND emitln
        S" 1 b# -> eax add, eax -> " name CHAR-APPEND  S"  dword-ptr mov," $+ emitln
        S" [esp] -> eax cmp," emitln
        L2 COUNT S"  offset NEAR  jg,"  $+ emitln
        L2 block
        'e' match
        L1 COUNT S"  offset NEAR  jmp," $+ emitln
        L2 COUNT postlabel
        S" [esp 4 +] -> esp lea," emitln
        L1   FREE ?ALLOCATE
        L2   FREE ?ALLOCATE
        name FREE ?ALLOCATE ;

-- recognize and translate a REPEAT construct
: doREPEAT ( -- )
        0 0 LOCALS| L2 L1 |
        'r' match
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L1 PACK DROP
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP
        L1 COUNT postlabel
        L2 block
        'u' match
        condition
        L1 COUNT S"  offset NEAR  je," $+ emitln
        L2 COUNT postlabel
        L1 FREE ?ALLOCATE
        L2 FREE ?ALLOCATE ;

-- recognize and translate a DO construct
: doDO ( -- )
        0 0 LOCALS| L2 L1 |
        'd' match
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L1 PACK DROP
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP
        expression
        L1 COUNT postlabel
        S" eax push,"              emitln
        L2 block
        'e' match
        S" eax pop,"               emitln
        S" 1 b# -> eax sub,"       emitln
        L1 COUNT
        S"  offset NEAR  jnz," $+  emitln
        S" [esp -4 +] -> esp lea," emitln
        L2 COUNT postlabel
        S" [esp  4 +] -> esp lea," emitln
        L1 FREE ?ALLOCATE
        L2 FREE ?ALLOCATE ;

-- recognize and translate a BREAK
: doBREAK ( label -- )
        'b' match
        DUP C@ 0= IF  S" No loop to break from" aborts  ENDIF
        COUNT S"  offset NEAR  jmp," $+ emitln ;

-- recognize and translate a statement block
:NONAME ( label -- )
        LOCAL L
        BEGIN   Look 'e' <>
                Look 'l' <> AND
```

```
          Look 'u' <> AND
   WHILE  CASE Look
          'i' OF  L doIF    ENDOF
          'w' OF  doWHILE   ENDOF
          'p' OF  doLOOP    ENDOF
          'r' OF  doREPEAT  ENDOF
          'f' OF  doFOR     ENDOF
          'd' OF  doDO      ENDOF
          'b' OF  L doBREAK ENDOF
                  other
         ENDCASE
   REPEAT ; IS block

-- parse and translate a program
: doprogram ( -- )
        C" " block
        Look 'e' <> IF  S" End" expected EXIT  ENDIF
        S" END" emitln ;

-- Main Program ----------------------------

: main ( -- ) CR init doprogram ;
```

6. Булевы выражения

ВВЕДЕНИЕ

В пятой части этой серии мы рассмотрели управляющие конструкции и разработали подпрограммы синтаксического анализа для трансляции их в объектный код. Мы закончили с хорошим, относительно богатым набором конструкций.

Однако, когда мы оставили синтаксический анализатор, в наших возможностях существовал один большой пробел: мы не обращались к вопросу условия ветвления. Чтобы заполнить пустоту, я представил вам фиктивную подпрограмму анализа Condition, которая служила только как заменитель настоящей.

Одним из дел, которыми мы займемся на этом уроке, будет заполнение этого пробела посредством расширения Condition до настоящего анализатора/транслятора.

ПЛАН

Мы собираемся подойти к этой главе немного по-другому, чем к любой другой. В других главах мы начинали немедленно с экспериментов, используя компилятор Pascal, выстраивая синтаксические анализаторы от самых элементарных начал до их конечных форм, не тратя слишком много времени на предварительное планирование. Это называется кодированием без спецификации и обычно к нему относятся неодобрительно. Раньше мы могли избегать планирования, потому что правила арифметики довольно хорошо установлены... мы знаем, что означает знак "+" без необходимости подробно это обсуждать. То же самое относится к ветвлениям и циклам. Но способы, которыми языки программирования реализуют логику, немного отличаются от языка к языку. Поэтому прежде, чем мы начнем серьезное кодирование, лучше мы сперва примем решение что же мы хотим. И способ сделать это находится на уровне синтаксических правил БНФ (грамматики).

ГРАММАТИКА

Некоторое время назад мы реализовали синтаксические уравнения БНФ для арифметических выражений фактически даже не записав их все в одном месте. Пришло время сделать это. Вот они:

```
<expression> ::= <unary op> <term> [<addop> <term>]*
<term>       ::= <factor> [<mulop> factor]*
<factor>     ::= <integer> | <variable> | ( <expression> )
```

(Запомните, преимущества этой грамматики в том, что она осуществляет такую иерархию приоритетов операторов, которую мы обычно ожидаем для алгебры.)

На самом деле, пока мы говорим об этом, я хотел бы прямо сейчас немного исправить эту грамматику. Способ, которым мы обрабатываем унарный минус, немного неудобный. Я нашел, что лучше записать грамматику таким образом:

```
<expression>    ::= <term> [<addop> <term>]*
<term>          ::= <signed factor> [<mulop> factor]*
<signed factor> ::= [<addop>] <factor>
<factor>        ::= <integer> | <variable> | (<expression>)
```

Это возлагает обработку унарного минуса на Factor, которому он в действительности и принадлежит.

Это не означает, что вы должны возвратиться назад и переписать программы, которые вы уже написали, хотя вы свободны сделать так, если хотите. Но с этого момента я буду использовать новый синтаксис.

Теперь, возможно, для вас не будет ударом узнать, что мы можем определить аналогичную грамматику для булевой алгебры. Типичный набор правил такой:

```
<b-expression>::= <b-term> [<orop> <b-term>]*
<b-term>      ::= <not-factor> [AND <not-factor>]*
<not-factor>  ::= [NOT] <b-factor>
<b-factor>    ::= <b-literal> | <b-variable> | (<b-expression>)
```

Заметьте, что в этой грамматике оператор AND аналогичен "*", а OR (и исключающее OR) - "+". Оператор NOT аналогичен унарному минусу. Эта иерархия не является абсолютным стандартом... некоторые языки, особенно Ada, обрабатывают все логические операторы как имеющие одинаковый уровень приоритета... но это кажется естественным.

Обратите также внимание на небольшое различие способов, которыми обрабатываются NOT и унарный минус. В алгебре унарный минус считается идущим со всем термом и поэтому никогда не появляется более одного раза в данном терме. Поэтому выражение вида:

```
a * -b
```

или еще хуже:

```
a - -b
```

не разрешены. В булевой алгебре наоборот, выражение

```
a AND NOT b
```

имеет точный смысл и показанный синтаксис учитывает это.

ОПЕРАТОРЫ ОТНОШЕНИЙ

Итак, предполагая что вы захотите принять грамматику, которую я показал здесь, мы теперь имеем синтаксические правила и для арифметики и для булевой алгебры. Сложность возникает когда мы должны объединить их. Почему мы должны сделать это? Ну, эта тема возникла из-за необходимости обрабатывать "предикаты" (условия), связанные с управляющими операторами такими как IF. Предикат должен иметь логическое значение, то есть он должен быть оценен как TRUE или FALSE. Затем ветвление выполняется или не выполняется в зависимости от этого значения. Тогда то, что мы ожидаем увидеть происходящим в процедуре Condition, будет вычисление булевого выражения.

Но имеется кое-что еще. Настоящее булево выражение может действительно быть предикатом управляющего оператора... подобно:

```
IF a AND NOT b THEN ....
```

Но более часто мы видим, что булева алгебра появляется в таком виде:

IF (x >= 0) and (x <= 100) THEN ...

Здесь два условия в скобках являются булевыми выражениями, но индивидуальные сравниваемые термы: x, 0 и 100 являются числовыми по своей природе. Операторы отношений >= и <= являются катализаторами, с помощью которых булевские и арифметические компоненты объединяются вместе.

Теперь, в примере выше сравниваемые термы являются просто термами. Однако, в общем случае, каждая сторона может быть математическим выражением. Поэтому мы можем определить отношение как:

```
<relation> ::= <expression> <relop> <expression>
```

где выражения, о которых мы говорим здесь - старого числового типа, а операторы отношений это любой из обычных символов:

```
=, <> (or !=), <, >, <=, and >=
```

Если вы подумаете об этом немного, то согласитесь, что так как этот вид предиката имеет логическое значение, TRUE или FALSE, это в действительности просто еще один вид показателя. Поэтому мы можем расширить определение булевого показателя следующим образом:

```
<b-factor> ::=    <b-literal>
                | <b-variable>
                | (<b-expression>)
                | <relation>
```

Вот эта связь! Операторы отношений и отношения, которые они определяют, служат для объединения двух типов алгебры. Нужно заметить, что это подразумевает иерархию, в которой арифметическое выражение имеет более высокий приоритет, чем булевский показатель и, следовательно, чем все булевы операторы. Если вы выпишите уровни приоритета для всех операторов, вы прийдете к следующему списку:

Уровень	Синтаксический элемент	Оператор
0	factor	literal, variable
1	signed factor	unary minus
2	term	*, /
3	expression	+, -
4	b-factor	literal, variable, relop
5	not-factor	NOT
6	b-term	AND
7	b-expression	OR, XOR

Если мы захотим принять столько уровней приоритета, эта грамматика кажется приемлемой. К несчастью, она не будет работать! Грамматика может быть великолепной в теории, но она может совсем не иметь смысла в практике нисходящего синтаксического анализатора. Чтобы увидеть проблему рассмотрите следующий фрагмент кода:

```
IF ((((((A + B + C) < 0 ) AND ....
```

Когда синтаксический анализатор анализирует этот код он знает, что после того, как он рассмотрит токен IF следующим должно быть булево выражение. Поэтому он может приступить к началу вычисления такого выражения. Но первое выражение в примере

является арифметическим выражением A + B + C. Хуже того, в точке, в которой анализатор прочитал значительную часть входной строки:

```
IF ((((((A   ,
```

он все еще не имеет способа узнать с каким видом выражения он имеет дело. Так не пойдет, потому что мы должны иметь две различных программы распознавания для этих двух случаев. Ситуация может быть обработана без изменения наших определений но только если мы захотим принять произвольное количество возвратов (backtracking) чтобы избавить наш путь от неверных предположений. Ни один из создателей компиляторов в здравом уме не согласился бы на это.

Происходит то, что красота и элегантность грамматики БНФ столкнулась лицом к лицу с реальностью технологии компиляции.

Чтобы работать с этой ситуацией создатели компиляторов должны идти на компромиссы, так чтобы один анализатор мог бы поддерживать грамматику без возвратов.

ИСПРАВЛЕНИЕ ГРАММАТИКИ

Проблема, с которой мы столкнулись, возникает потому, что наше определение и арифметических и булевых показателей позволяет использовать выражения в скобках. Так как определения рекурсивны, мы можем закончить с любым числом уровней скобок и синтаксический анализатор не может знать с каким видом выражения он имеет дело.

Решение просто, хотя и приводит к глубоким изменениям нашей грамматики. Мы можем разрешить круглые скобки только в одном виде показателей. Способ сделать это значительно изменяется от языка к языку. Это то место, где не существует соглашения или договора способного нам помочь.

Когда Никлаус Вирт разработал Паскаль, его желанием было ограничить количество уровней приоритета (меньше подпрограмм синтаксического анализа, в конце концов). Так операторы OR и исключающее OR рассматриваются просто как Addop и обрабатываются на уровне математического выражения. Аналогично AND рассматривается подобно Mulop и обрабатывается с Term. Уровни приоритета:

Уровень	Синтаксический элемент	Оператор
0	factor	literal, variable
1	signed factor	unary minus, NOT
2	term	*, /, AND
3	expression	+, -, OR

Заметьте, что имеется только один набор синтаксических правил, применимый к обоим видам операторов. Тогда согласно этой грамматике выражения типа:

```
x + (y AND NOT z) DIV 3
```

являются совершенно допустимыми. И, фактически, они таковыми являются... настолько, насколько синтаксический анализатор в этом заинтересован. Паскаль не позволяет смешивать арифметические и логические переменные, и подобные вещи скорее перехватываются на семантическом уровне, когда придет время генерировать для них код, чем на синтаксическом уровне.

Авторы C взяли диаметрально противоположный метод: они обрабатывают операторы как разные и C имеет что-то гораздо более похожее на наши семь уровней приоритета. Фактически, в C имеется не менее 17 уровней! Дело в том, что C имеет также операторы '=', '+=' и их родственников '<<', '>>', '++', '--' и т.д. Как ни странно, хотя в C арифметические и булевые операторы обрабатываются раздельно, то переменные нет... в C нет никаких булевых или логических переменных, так что логическая проверка может быть сделана на любом

целочисленном значении.

Мы сделаем нечто среднее. Я склонен обычно придерживаться Паскалевского подхода, так как он кажется самым простым с точки зрения реализации, но это приводит к некоторым странностям, которые я никогда очень сильно не любил, как например в выражении:

IF (c >= 'A') and (c <= 'Z') then ...

скобки обязательны. Я никогда не мог понять раньше почему, и ни мой компилятор, ни любой человек также не объясняли этого достаточно хорошо. Но сейчас мы все можем видеть, что оператор "and", имеющий приоритет как у оператора умножения, имеет более высокий приоритет, чем у операторов отношения, поэтому без скобок выражение эквивалентно:

```
IF c >= ('A' and c) <= 'Z' then
```

что не имеет смысла.

В любом случае, я решил разделить операторы на различные уровни, хотя и не столько много как в C.

```
<b-expression> ::= <b-term> [<orop> <b-term>]*
<b-term>       ::= <not-factor> [AND <not-factor>]*
<not-factor>   ::= [NOT] <b-factor>
<b-factor>     ::= <b-literal> | <b-variable> | <relation>
<relation>     ::= | <expression> [<relop> <expression]
<expression>   ::= <term> [<addop> <term>]*
<term>         ::= <signed factor> [<mulop> factor]*
<signed factor>::= [<addop>] <factor>
<factor>       ::= <integer> | <variable> | (<b-expression>)
```

Эта грамматика приводит к тому же самому набору семи уровней, которые я показал ранее. Действительно, это почти таже самая грамматика... я просто исключил заключенное в скобки b-выражение как возможный b-показатель и добавил отношение как допустимую форму b-показателя.

Есть одно тонкое, но определяющее различие, которое заставляет все это работать. Обратите внимание на квадратные скобки в определении отношения. Это означает, что relop и второе выражение являются необязательными. Странным последствием этой грамматики (которое присутствует и в C) является то, что каждое выражения потенциально является булевым выражение. Синтаксический анализатор всегда будет искать булевское выражение, но "уладит" все до арифметического. Честно говоря, это будет замедлять синтаксический анализатор потому что он должен пройти через большее количество вызовов процедур. Это одна из причин, почему компиляторы Паскаля обычно быстрее выполяют компиляцию, чем компиляторы C. Если скорость для вас - больное место, придерживайтесь синтаксиса Паскаля.

СИНТАКСИЧЕСКИЙ АНАЛИЗАТОР

Теперь, когда мы прошли через процесс принятия решений, мы можем поспешить с разработкой синтаксического анализатора. Вы делали это со мной несколько раз до этого, поэтому вы знаете последовательность: мы начнем с новой копии Cradle и будем добавлять процедуры одна за другой. Так что давайте сделаем это.

Мы начинаем, как и в случае с арифметикой, работая с булевыми литералами а не с переменными. Это дает нам новый вид входного токена, поэтому нам также нужна новая программа распознавания и новая процедура для чтения экземпляров этого типа токенов. Давайте начнем, определив эти две новые процедуры:

```
--  -------------------------------------------------------------
```

```
-- recognize a boolean literal
: boolean? ( char -- tf )
        >UPC DUP 'T' = SWAP 'F' = OR ;
-- ----------------------------------------------------------------
-- get a boolean literal
: getboolean ( -- flag )
        Look boolean? 0= IF  S" Boolean literal" expected  ENDIF
        Look >UPC 'T' =
        getchar ;
-- ----------------------------------------------------------------
```

Внесите эти подпрограммы в вашу программу. Вы можете протестировать их, добавив в основную программу оператор печати:

WriteLn(GetBoolean);

Откомпилируйте программу и протестируйте ее(chap6a.frt). Как обычно пока не очень впечатляет но скоро будет.

Теперь, когда мы работали с числовыми данными, мы должны были организовать генерацию кода для загрузки значений в D0. Нам необходимо сделать то же самое и для булевых данных. Обычным способом кодирования булевых переменных является использование 0 для представления FALSE и какого-либо другого значения для TRUE. Многие языки, как например C, используют для его представления целое число 1. Но я предпочитаю FFFF (или -1) потому что побитовое NOT также возвратит логическое NOT. Итак, нам теперь нужно выдать правильный ассемблерный код для загрузки этих значений. Первая засечка на синтаксическом анализаторе булевых выражений (BoolExpression, конечно):

```
-- ----------------------------------------------------------------
-- Parse and translate a boolean expression
: boolexpression ( -- )
        Look boolean? 0= IF  S" Boolean literal" expected  ENDIF
        getboolean IF  S" -1 b# -> eax  mov,"
                 ELSE  S" eax -> eax  xor,"
                ENDIF  emitln ;
-- ----------------------------------------------------------------
```

Добавьте эту процедуру в ваш анализатор и вызовите ее из основной программы (заменив оператор печати, который вы только что там поместили). Как вы можете видеть, мы все еще не имеем значительной части синтаксического анализатора, но выходной код начинает выглядеть более реалистичным. (chap6b.frt).

Затем, конечно, мы должны расширить определение булевого выражения. У нас уже есть правило в БНФ:

```
 <b-expression> ::= <b-term> [<orop> <b-term>]*
```

Я предпочитаю Паскалевскую версию "orop" - OR и XOR. Но так как мы сохраняем односимвольные токены, я буду кодировать их знаками '|' и '~'. Следующая версия BoolExpression - почти полная копия арифметической процедуры Expression:

```
-- ---------------------------------------------------------------
-- Recognize and translate a boolean or
: boolOR ( -- )
        '|' match
        boolterm
        S" [esp] -> eax or, [esp 4 +] -> esp lea," emitln ;
-- ---------------------------------------------------------------
-- Recognize and translate an exclusive or
: boolXOR ( -- )
        '~' match
        boolterm
```

```
        S" [esp] -> eax xor, [esp 4 +] -> esp lea," emitln ;
-- ------------------------------------------------------------
-- Parse and translate a boolean expression
: boolexpression ( -- )
        boolterm
        BEGIN   Look orop?
        WHILE   S" eax push," emitln
                CASE Look
                  '|' OF  boolOR  ENDOF
                  '~' OF  boolXOR ENDOF
                ENDCASE
        REPEAT ;
-- -------------------------------------------------------------
```

Обратите внимание на новую процедуру OrOp?, которая также является копией, на этот раз IsAddOp:

```
-- ------------------------------------------------------------
-- Recognize a boolean orop
: orop? ( char -- tf )  DUP '|' = SWAP '~' = OR ;
-- ------------------------------------------------------------
```

OK, переименуйте старую версию BoolExpression в BoolTerm, затем наберите код, представленный выше. Откомпилируйте и протестируйте эту версию. К этому моменту выходной код начинает выглядеть довольно хорошим. Конечно, нет большого смысла от булевой алгебры над постоянными значениями, но скоро мы расширим булевы типы, с которыми мы работаем. (chap6c.frt).

Возможно вы уже предположили, какой будет следующий шаг: булевская версия Term.

Переименуйте текущую процедуру BoolTerm в NotFactor, и введите следующую новую версию BoolTerm. Заметьте, что она намного более простая, чем числовая версия, так как здесь нет эквивалента деления.

```
-- ------------------------------------------------------------
-- Parse and translate a boolean term
: boolterm ( -- )
        notfactor
        BEGIN   Look '&' =
        WHILE   S" eax push," emitln
                '&' match
                notfactor
                S" [esp] -> eax  and,  [esp 4 +] -> esp lea," emitln
        REPEAT ;
-- ------------------------------------------------------------
```

Теперь мы почти дома. Мы транслируем сложные булевые выражения, хотя только и для постоянных значений. Следующий шаг - учесть NOT. Напишите следующую процедуру:

```
-- ------------------------------------------------------------
-- parse and translate a boolean factor with NOT
: notfactor ( -- )
        Look '!' <> IF  boolfactor EXIT  ENDIF
        '!' match
        boolfactor
        S" -1 b# -> eax  xor," emitln ;
-- ------------------------------------------------------------
```

И переименуйте предыдущую процедуру в BoolFactor. Теперь испытайте компилятор. К этому времени синтаксический анализатор должен обрабатывать любое булевое выражение, которое вы позаботитесь ему подкинуть. Работает? Отлавливает ли он неправильно сформированные выражения?

Если вы следили за тем, что мы делали в синтаксическом анализаторе для

математических выражений вы знаете что далее мы расширили определение показателя для включения переменных и круглых скобок. Мы не должны делать это для булевого показателя, потому что об этих маленьких вещах позаботится наш следующий шаг. Необходима только одна дополнительная строка в BoolFactor, чтобы позаботиться об отношениях:

```
-- --------------------------------------------------------------
-- Parse and translate a boolean factor
: boolfactor ( -- )
        Look boolean?
           IF getboolean IF  S" -1 b# -> eax  mov,"
                       ELSE  S" eax -> eax  xor,"
                      ENDIF  emitln
         ELSE relation
        ENDIF ;
-- --------------------------------------------------------------
```

Вы могли бы задаться вопросом, когда я собираюсь предоставить булевские переменные и булевские выражения в скобках. Отвечаю: никогда. Помните, ранее мы убрали их из грамматики. Прямо сейчас я собираюсь кодировать грамматику, которую мы уже согласовали. Сам компилятор не может видеть разницы между булевыми переменными или выражениями и арифметическими переменными или выражениями... все это будет обрабатываться в Relation в любом случае.

Конечно, понадобится некоторый код для Relation. Однако, я не чувствую себя комфортно, добавляя еще код, не проверив сперва тот, который мы уже имеем. Так что давайте сейчас просто напишем фиктивную версию Relation, которая ничего не делает за исключением того, что съедает текущий символ и выводит небольшое сообщение:

```
-- ---------------------------------------------------------------
-- Parse and translate a relation
: relation ( -- ) S" <Relation>" emitln getchar ;
-- ---------------------------------------------------------------
```

OK, наберите этот код и испытайте его. Все старые дела все еще должны работать... у вас должна быть возможность генерировать код для AND, OR и NOT. Кроме того, если вы наберете любой алфавитный символ, вы должны получить небольшой заменитель <Relation>, где должен быть булев показатель. Вы получили это? Отлично, тогда давайте перейдем к полной версии Relation.(chap6e.frt).

Чтобы получить ее, тем не менее, сначала мы должны положить небольшое основание. Вспомните, что отношение имеет форму:

```
 <relation>     ::= | <expression> [<relop> <expression]
```

Так как у нас появился новый вид операторов, нам также понадобится новая логическая функция для ее распознавания. Эта функция показана ниже. Из-за ограничения в один символ, я придерживаюсь четырех операторов, которые могут быть закодированы такими символами ("не равно" закодировано как "#").

```
-- --------------------------------------------------------------
-- Recognize a relop
: relop? ( char -- tf )
        DUP  '=' =
        OVER '#' = OR
        OVER '<' = OR
        SWAP '>' = OR ;
-- --------------------------------------------------------------
```

1 в регистре D0 для представления логического значения и также то, что операторы цикла ожидают, что будет установлен соответствующий флаг. При реализации всего этого для

68000 все становится немного сложным.

Так как операторы цикла выполняются только по флажкам, было бы хорошо (а также довольно эффективно) просто установить эти флажки и совсем ничего не загружать в D0. Это было бы прекрасно для циклов и ветвлений, но запомните, что отношения могут быть использованы везде, где могут быть использованы булевы показатели. Мы можем сохранять его результат в булевой переменной. Так как мы не можем знаеть пока как будет использоваться результат, мы должны учесть оба случая.

Сравнение числовых данных достаточно просто... 68000 имеет команду для этого... но она устанавливает флажки а не значение. Более того, всегда будут устанавливаться те же самые флажки (ноль если равно, и т.д.), в то время, как нам необходим по-разному установленный флажок нуля для каждого различного оператора отношения.

Решение заключается в инструкции Scc процессора 68000, которая устанавливает значение байта в 0000 или FFFF (забавно как это работает!) в зависимости от результата определенного условия. Если мы сделаем байтом результата регистр D0, мы получим необходимое логическое значение.

К сожалению, имеется одно заключительное осложнение: в отличие от почти всех других команд в наборе 68000, Scc не сбрасывает флажки условий в соответствии с сохраняемыми данными. Поэтому мы должны сделать последний шаг, проверить D0 и установить соответствующим образом флажки. Это должно быть похоже на оборот вокруг луны для получения того, что мы хотим: мы сначала выполняем проверку, затем проверяем флажки, чтобы установить данные в D0, затем тестируем D0 чтобы установить флажки снова. Это окольный путь, но это самый простой способ получить правильные флажки и, в конце концов, это всего лишь пара инструкций.

Я мог бы упомянуть здесь, что эта область, по моему мнению, показывает самые большие различия между эффективностью вручную написанного на ассемблере и сгенерированного компилятором кода. Мы уже видели, что мы теряем эффективность при арифметических операциях, хотя позже я планирую показать вам как ее немного улучшить. Мы также видели, что управляющие конструкции сами по себе могут быть реализованы довольно эффективно... обычно очень сложно улучшить код, сгенерированный для IF или WHILE. Но практически каждый компилятор, который я когда-либо видел, генерирует ужасный код, по сравнению с ассемблером, для вычисления булевых функций и особенно отношений. Причина как раз в том, о чем я упомянул выше. Когда я пишу код на ассемблере, я двигаюсь вперед и выполняю проверку наиболее удобным для меня способом, и затем подготавливаю ветвление так чтобы переход был выполнен на нужную ветку. Фактически, я "подстраиваю" каждое ветвление под ситуацию. Компилятор не может сделать этого (практически) и он также не может знать, что нам не нужно сохранять результат проверки как булевскую переменную. Поэтому он должен генерировать код по очень строгим правилам и часто заканчивает сохранением результата как булевой переменной, которая никогда не будет использована для чего-либо.

В любом случае мы теперь готовы рассмотреть код для Relation. Он показан ниже с сопровождающими процедурами:

```
-- -----------------------------------------------------------------------------
-- recognize and translate a relational "equals"
: equals ( -- )
  '=' match
  expression
  S" [esp] -> eax  cmp,  [esp 4 +] -> esp  lea,  al  sete,  al -> eax  movsx,"
emitln ;
-- -----------------------------------------------------------------------------
-- recognize and translate a relational "not equals"
: notequals ( -- )
  '#' match
  expression
  S" [esp] -> eax  cmp,  [esp 4 +] -> esp  lea,  al  setne, al -> eax  movsx,"
emitln ;
-- -----------------------------------------------------------------------------
```

```
-- recognize and translate a relational "less than"
: less ( -- )
        '<' match
        expression
        S" [esp] -> eax  cmp,  [esp 4 +] -> esp  lea,  al  setg, al -> eax  movsx,"
emitln ;
--
---------------------------------------------------------------------------------
-- recognize and translate a relational "greater than"
: greater ( -- )
        '>' match
        expression
        S" [esp] -> eax  cmp,  [esp 4 +] -> esp  lea,  al  setl, al -> eax  movsx,"
emitln ;
--
---------------------------------------------------------------------------------
-- parse and translate a relation
: relation ( -- )
        expression
        Look relop? 0= ?EXIT
        S" eax push," emitln
        CASE Look
          '=' OF  equals     ENDOF
          '#' OF  notequals  ENDOF
          '<' OF  less       ENDOF
          '>' OF  greater    ENDOF
        ENDCASE ;
-- ---------------------------------------------------------------
```

Теперь этот вызов Expression выглядит знакомым! Вот где редактор вашей системы оказывается полезным. Мы уже генерировали код для Expression и его близнецов на предыдущих уроках. Теперь вы можете скопировать их в ваш файл. Не забудьте использовать односимвольную версию. Просто чтобы быть уверенным, я продублировал арифметические процедуры ниже. Если вы наблюдательны, вы также увидите, что я их немного изменил чтобы привести в соответствие с последней версией синтаксиса. Эти изменения не являются необходимыми, так что вы можете предпочесть оставить все как есть до тех пор, пока не будете уверены, что все работает.

```
-- ---------------------------------------------------------------
-- getname still returns single characters
VARIABLE gn$ : getname ( -- c-addr u ) getname gn$ C! gn$ 1 ;
-- ---------------------------------------------------------------
-- Parse and translate an identifier
: ident ( -- )
        getname DUP 1+ ALLOCATE ?ALLOCATE LOCAL name
        name PACK DROP
        Look '(' = IF   '(' match ')' match
                        name COUNT S"  offset NEAR  call,"
                 ELSE   name COUNT S"  dword-ptr -> eax  mov,"
                ENDIF   $+ emitln
        name FREE ?ALLOCATE ;
-- ---------------------------------------------------------------

DEFER expression

-- ---------------------------------------------------------------
-- parse and translate a math factor
: factor ( -- )
        Look '('  = IF  '(' match  expression  ')' match EXIT
                 ENDIF
        Look alpha? IF  ident EXIT  ENDIF
        getnum S"  d# -> eax  mov," $+ emitln ;
-- ---------------------------------------------------------------
-- parse and translate the first math factor
: signedfactor ( -- )
```

```
        Look '+'  = IF  getchar       ENDIF
        Look '-' <> IF  factor  EXIT  ENDIF
        getchar
        Look digit? IF  S" #-" getnum $+ S"  d# -> eax  mov," $+
                  ELSE  factor S" eax  neg,"
                 ENDIF  emitln ;
-- ----------------------------------------------------------------
-- recognize and translate multiply and divide
: multiply  ( -- )
        '*' match  factor
        S" [esp] dword  mul, [esp 4 +] -> esp  lea," emitln ;
-- ----------------------------------------------------------------
: divide ( -- )
        '/' match  factor
        S" ebx pop, ebx -> eax  xchg,  eax -> edx  mov," emitln
        S" #31 b# -> edx  sar,  ebx  idiv," emitln ;
-- ----------------------------------------------------------------
-- parse and translate a math term
: term ( -- )
        signedfactor
        BEGIN Look mulop?
        WHILE   S" eax push," emitln
                CASE Look
                  '*' OF multiply  ENDOF
                  '/' OF divide    ENDOF
                         S" Mulop" expected
                ENDCASE
        REPEAT ;
-- ----------------------------------------------------------------
-- recognize and translate add and subtract
: add ( -- )
        '+' match  term
        S" [esp] -> eax  add,  [esp 4 +] -> esp  lea," emitln ;
-- ----------------------------------------------------------------
: subtract ( -- )
        '-' match  term
        S" [esp] -> eax  sub,  [esp 4 +] -> esp  lea," emitln
        S" eax  neg," emitln ;
-- ----------------------------------------------------------------
-- parse and translate an expression
:NONAME ( -- )
        term
        BEGIN   Look addop?
        WHILE   S" eax push," emitln
                CASE Look
                  '+' OF add      ENDOF
                  '-' OF subtract ENDOF
                         S" Addop" expected
                ENDCASE
        REPEAT ; IS expression
-- ----------------------------------------------------------------
```

Теперь вы получили что-то… синтаксический анализатор, который может обрабатывать и арифметику и булеву алгебру и их комбинации через использование операторов отношений. Я советую вам сохранить копию этого синтаксического анализатора в безопасном месте для будущих обращений, потому что на нашем следующем шаге мы собираемся разделить его.

ОБЪЕДИНЕНИЕ С УПРАВЛЯЮЩИМИ КОНСТРУКЦИЯМИ

Сейчас давайте возвратимся назад к файлу который мы создали ранее и который выполняет синтаксический анализ управляющих конструкций. Помните небольшие фиктивные процедуры Condition и Expression? Теперь вы знаете, что в них должно находиться!

Я предупреждаю вас, вы собираетесь сделать некоторые творческие изменения, поэтому

потратьте ваше время и сделайте это правильно. Вы должны скопировать все процедуры из анализатора логики от Ident до BoolExpression в синтаксический анализатор управляющих конструкций. Вставьте их в текущей позиции Condition. Затем удалите эту процедуру, так же как и фиктивную Expression. Затем замените каждый вызов Condition на обращение к BoolExpression. Наконец скопируйте процедуры IsMulop, IsOrOp, IsRelop, IsBoolean, и GetBoolean на место. Этого достаточно.

Откомпилируйте полученную программу и протестируйте ее. Так как мы не использовали эту программу некоторое время, не забудьте, что мы использовали односимвольные токены для IF, WHILE и т.д. Также не забудьте, что любая буква, не являющаяся ключевым словом, просто отображается на экране как блок.

Попробуйте:

```
ia=bxlye
```

что означает "IF a=b X ELSE Y ENDIF".

Что вы думаете? Работает? Попробуйте что-нибудь еще.

ДОБАВЛЕНИЕ ПРИСВАИВАНИЙ

Раз у нас уже есть подпрограммы для выражений, мы могли бы также заменить "блоки" настоящими операциями присваивания. Мы уже делали это прежде, поэтому это не будет слишком трудно. Прежде, чем сделать этот шаг, однако, мы должны исправить кое-что еще.

Скоро мы обнаружим, что наши однострочные "программы", которые мы здесь пишем, будут ограничивать наш стиль. В настоящее время у нас нет способа вылечить это, потому что наш компилятор не распознает символы конца строки, возврат каретки (CR) и перевод строки (LF). Поэтому перед продвижением дальше давайте заткнем эту дыру.

Существует пара способов для работы с CR/LF. Один (подход C/Unix) просто рассматривает их как дополнительные символы пробела и игнорирует их. Фактически это не такой плохой подход, но он приводит к странным результатам для нашего анализатора в его текущем состоянии. Если бы он считывал входной поток из исходного файла как любой уважающий себя настоящий компилятор, не было бы никаких проблем. Но мы считываем входной поток с клавиатуры и ожидаем, что должно что-то произойти, когда мы нажимаем клавишу Return. Этого не произойдет, если мы просто перескакиваем CR и LF (попробуйте это). Поэтому я собираюсь использовать здесь другой метод, который в конечном счете не обязательно является лучшим методом. Рассматривайте его как временную замену до тех пор, пока мы не двинемся дальше.

Вместо того, чтобы пропускать CR/LF, мы позволим синтаксическому анализатору двигаться вперед и отлавливать их, затем предоставлять их специальной процедуре, аналогичной SkipWhite, которая пропускает их только в определенных "допустимых" местах.

Вот эта процедура:

```
-- ------------------------------------------------------------
-- skip cr+lf
: fin ( -- )
        Look ^M = IF getchar ENDIF
        Look ^J = IF getchar ENDIF ;
-- ------------------------------------------------------------
```

Теперь добавьте два вызова Fin в процедуру Block следующим образом:

```
-- ------------------------------------------------------------
-- Recognize and translate a statement block
:NONAME ( label -- )
        LOCAL L
        BEGIN  Look 'e' <>
```

```
            Look 'l' <> AND
            Look 'u' <> AND
     WHILE  fin
            CASE Look
             'i' OF  L doIF    ENDOF
             'w' OF  doWHILE   ENDOF
             'p' OF  doLOOP    ENDOF
             'r' OF  doREPEAT  ENDOF
             'f' OF  doFOR     ENDOF
             'd' OF  doDO      ENDOF
             'b' OF  L doBREAK ENDOF
                     other
            ENDCASE
            fin
     REPEAT ; IS block
-- ---------------------------------------------------------------
```

Теперь вы обнаружите, что можете использовать многострочные "программы". Единственное ограничение в том, что вы не можете отделять токены IF или WHILE от их предикатов.

Теперь мы готовы включить операторы присваивания. Просто замените вызов Other в процедуре Block на вызов Assignment и добавьте следующую процедуру, скопированную из одной нашей более ранней программы. Обратите внимание, что сейчас Assignment вызывает BoolExpression, поэтому мы можем присваивать логические переменные.

```
-- ---------------------------------------------------------------
-- Parse and translate an assignment statement
: assignment ( -- )
   getname DUP 1+ ALLOCATE ?ALLOCATE DUP LOCAL name PACK DROP
   '=' match  boolexpression
   S" eax -> " name COUNT $+ S"  dword-ptr  mov," $+ emitln
   name FREE ?ALLOCATE ;
-- ---------------------------------------------------------------
```

С этими изменениями у вас теперь должна быть возможность писать сносные, реалистично выглядящие программы, подчиненные только нашему ограничению односимвольными токенами. Первоначально я также намеревался избавить вас и от этого ограничения. Однако, это потребует довольно больших изменений того, что мы сделали к этому моменту. Нам нужен настоящий лексический анализатор и это требует некоторых структурных изменений. Это небольшие изменения, которые потребуют чтобы мы выбросили все, что мы сделали к этому времени... при желании это может быть сделано в действительности с минимальными изменениями. Но необходимо такое желание.

Эта глава и так получилась довольно длинной и она содержит довольно тяжелый материал, поэтому я решил оставить этот шаг до следующего раза, чтобы у вас было немного времени усвоить то, что мы сделали и вы были готовы начать на свежую голову.

В следующей главе, мы построим лексический анализатор и устраним односимвольный барьер раз и навсегда. Мы также напишем наш первый законченный компилятор, основанный на том, что мы сделали на этом уроке. Увидимся.

7. Лексический анализ

ВВЕДЕНИЕ

В последней главе я оставил вас с компилятором который должен почти работать, за исключением того, что мы все еще ограничены односимвольными токенами. Цель этого урока состоит в том, чтобы избавиться от этого ограничения раз и навсегда. Это означает, что мы должны иметь дело с концепцией лексического анализатора (сканера).

Возможно я должен упомянуть, почему нам вообще нужен лексический анализатор... в конце концов до настоящего времени мы были способны хорошо справляться и без него даже когда мы предусмотрели многосимвольные токены.

Единственная причина, на самом деле, имеет отношение к ключевым словам. Это факт компьютерной жизни, что синтаксис ключевого слова имеет ту же самую форму, что и синтаксис любого другого идентификатора. Мы не можем сказать пока не получим полное слово действительно ли это ключевое слово. К примеру переменная IFILE и ключевое слово IF выглядят просто одинаковыми до тех пор, пока вы не получите третий символ. В примерах до настоящего времени мы были всегда способны принять решение, основанное на первом символе токена, но это больше невозможно когда присутствуют ключевые слова. Нам необходимо знать, что данная строка является ключевым словом до того, как мы начнем ее обрабатывать. И именно поэтому нам нужен сканер.

На последнем уроке я также пообещал, что мы могли бы предусмотреть нормальные токены без глобальных изменений того, что мы уже сделали. Я не солгал... мы можем, как вы увидите позднее. Но каждый раз, когда я намеревался встроить эти элементы в синтаксический анализатор, который мы уже построили, у меня возникали плохие чувства в отношении их. Все это слишком походило на временную меру. В конце концов я выяснил причину проблемы: я установил программу лексического анализа не объяснив вам вначале все о лексическом анализе, и какие есть альтернативы. До настоящего времени я старательно избегал давать вам много теории и, конечно, альтернативные варианты. Я обычно не воспринимаю хорошо учебники которые дают двадцать пять различных способов сделать что-то, но никаких сведений о том, какой способ лучше всего вам подходит. Я попытался избежать этой ловушки, просто показав вам один способ, который работает.

Но это важная область. Хотя лексический анализатор едва ли является наиболее захватывающей частью компилятора он часто имеет наиболее глубокое влияние на общее восприятие языка так как эта часть наиболее близка пользователю. Я придумал специфическую структуру сканера, который будет использоваться с KISS. Она соответствует восприятию, которое я хочу от этого языка. Но она может совсем не работать для языка, который придумаете вы, поэтому в этом единственном случае я чувствую, что вам важно знать ваши возможности.

Поэтому я собираюсь снова отклониться от своего обычного распорядка. На этом уроке мы заберемся гораздо глубже, чем обычно, в базовую теорию языков и грамматик. Я также буду говорить о других областях кроме компиляторов в которых лексических анализ играет важную роль. В заключение я покажу вам некоторые альтернативы для структуры лексического анализатора. Тогда и только тогда мы возвратимся к нашему синтаксическому анализатору из последней главы. Потерпите... я думаю вы найдете, что это стоит ожидания. Фактически, так как сканеры имеют множество применений вне компиляторов, вы сможете легко убедиться, что это будет наиболее полезный для вас урок.

ЛЕКСИЧЕСКИЙ АНАЛИЗ

Лексический анализ - это процесс сканирования потока входных символов и разделения его на строки, называемые лексемами. Большинство книг по компиляторам начинаются с этого и посвящают несколько глав обсуждению различных методов построения сканеров. Такой подход имеет свое место, но, как вы уже видели, существуют множество вещей, которые вы можете сделать даже никогда не обращавшись к этому вопросу, и, фактически, сканер, который мы здесь закончим, не очень будет напоминать то, что эти тексты описывают. Причина? Теория компиляторов и, следовательно, программы следующие из нее, должны работать с большинством общих правил синтаксического анализа. Мы же не делаем этого. В реальном мире возможно определить синтаксис языка таким образом, что будет достаточно довольно простого сканера. И как всегда KISS - наш девиз.

Как правило, лексический анализатор создается как отдельная часть компилятора, так что синтаксический анализатор по существу видит только поток входных лексем. Теоретически нет необходимости отделять эту функцию от остальной части синтаксического

анализатора. Имеется только один набор синтаксических уравнений, который определяет весь язык, поэтому теоретически мы могли бы написать весь анализатор в одном модуле.

Зачем необходимо разделение? Ответ имеет и теоретическую и практическую основы.

В 1956 Ноам Хомский определил "Иерархию Хомского" для грамматик. Вот они:

- Тип 0. Неограниченные (например Английский язык)
- Тип 1. Контекстно-зависимые
- Тип 2. Контекстно-свободные
- Тип 3. Регулярные.

Некоторые характеристики типичных языков программирования (особенно старых, таких как Фортран) относят их к Типу 1, но большая часть всех современных языков программирования может быть описана с использованием только двух последних типов и с ними мы и будем здесь работать.

Хорошая сторона этих двух типов в том, что существуют очень специфические пути для их анализа. Было показано, что любая регулярная грамматика может быть анализирована с использованием частной формы абстрактной машины, называемой конечным автоматом. Мы уже реализовывали конечные автоматы в некоторых их наших распознающих программ.

Аналогично грамматики Типа 2 (контекстно-свободные) всегда могут быть анализированы с использованием магазинного автомата (конечный автомат, дополненный стеком). Мы также реализовывали эти машины. Вместо реализации явного стека для выполнения работы мы положились на встроенный стек связанный с рекурсивным кодированием и это фактически является предпочтительным способом для нисходящего синтаксического анализа.

Случается что в реальных, практических грамматиках части, которые квалифицируются как регулярные выражения, имеют склонность быть низкоуровневыми частями, как определение идентификатора:

```
<ident> ::= <letter> [ <letter> | <digit> ]*
```

Так как требуется различные виды абстрактных машин для анализа этих двух типов грамматик, есть смысл отделить эти низкоуровневые функции в отдельный модуль, лексический анализатор, который строится на идее конечного автомата. Идея состоит в том, чтобы использовать самый простой метод синтаксического анализа, необходимый для работы.

Имеется другая, более практическая причина для отделения сканера от синтаксического анализатора. Мы хотим думать о входном исходном файле как потоке символов, которые мы обрабатываем справа налево без возвратов. На практике это невозможно. Почти каждый язык имеет некоторые ключевые слова типа IF, WHILE и END. Как я упомянул ранее, в действительности мы не можем знать является ли данная строка ключевым словом до тех пор пока мы не достигнем ее конца, что определено пробелом или другим разделителем. Так что мы должны хранить строку достаточно долго для того, чтобы выяснить имеем мы ключевое слово или нет. Это ограниченная форма перебора с возвратом.

Поэтому структура стандартного компилятора включает разбиение функций низкоуровневого и высокоуровневого синтаксического анализа. Лексический анализатор работает на символьном уровне собирая символы в строки и т.п., и передавая их синтаксическому анализатору как неделимые лексемы. Также считается нормальным позволить сканеру выполнять работу по идентификации ключевых слов.

КОНЕЧНЫЕ АВТОМАТЫ И АЛЬТЕРНАТИВЫ

Я упомянул, что регулярные выражения могут анализироваться с использованием конечного автомата. В большинстве книг по компиляторам а также в большинстве компиляторов, вы обнаружите, что это применяется буквально. Обычно они имеют настоящую реализацию конечного автомата с целыми числами, используемыми для

определения текущего состояния и таблицей действий, выполняемых для каждой комбинации текущего состояния и входного символа. Если вы пишите "front end" для компилятора, используя популярные Unix инструменты LEX и YACC, это то, что вы получите. Выход LEX - конечый автомат, реализованный на C плюс таблица действий, соответствующая входной грамматике данной LEX. Вывод YACC аналогичен... исскуственный таблично-управляемый синтаксический анализатор плюс таблица, соответствующая синтаксису языка.

Однако это не единственный вариант. В наших предыдущих главах вы много раз видели, что возможно реализовать синтаксические анализаторы специально не имея дела с таблицами, стеками и переменными состояния. Фактически в пятой главе я предупредил вас, что если вы считает себя нуждающимся в этих вещах, возможно вы делаете что-то неправильно и не используете возможности Паскаля. Существует в основном два способа определить состояние конечного автомата: явно, с номером или кодом состояния и неявно, просто на основании того факта, что я нахожусь в каком-то определенном месте кода (если сегодня вторник, то это должно быть Бельгия). Ранее мы полагались в основном на неявные методы, и я думаю вы согласитесь, что они работают здесь хорошо.

На практике может быть даже не обязательно иметь четко определенный лексический анализатор. Это не первый наш опыт работы с многосимвольными токенами. В третьей главе мы расширили наш синтаксический анализатор для их поддержки и нам даже не был нужен лексический анализатор. Причиной было то, что в узком контексте мы всегда могли сказать просто рассматривая единственный предсказывающий символ, имеем ли мы дело с цифрой, переменной или оператором. В действительности мы построили распределенный лексический анализатор, используя процедуры GetName и GetNum.

Имея ключевые слов мы не можем больше знать с чем мы имеем дело до тех пор, пока весь токен не будет прочитан. Это ведет нас к более локализованному сканеру, хотя, как вы увидите, идея распределенного сканера все же имеет свои достоинства.

ЭКСПЕРИМЕНТЫ ПО СКАНИРОВАНИЮ

Прежде чем возвратиться к нашему компилятору, было бы полезно немного поэкспериментировать с общими понятиями.

Давайте начнем с двух определений, наиболее часто встречающихся в настоящих языках программирования:

```
<ident> ::= <letter> [ <letter> | <digit> ]*
<number ::= [<digit>]+
```

(Не забудьте, что "*" указывает на ноль или более повторений условия в квадратных скобках, а "+" на одно и более.)

Мы уже работали с подобными элементами в третьей главе. Давайте начнем (как обычно) с пустого Cradle. Не удивительно, что нам понадобится новая процедура распознавания:

```
-- ------------------------------------------------------------
-- Recognize an alphanumeric character
: alnum? ( char -- tf ) DUP alpha? SWAP digit? OR ;
-- ------------------------------------------------------------
```

Используя ее, давайте напишем следующие две подпрограммы, которые очень похожи на те, которые мы использовали раньше:

```
-- ------------------------------------------------------------
-- Get an identifier.
: char+! ( c addr -- ) DUP >R COUNT + C!  1 R> C+! ;

CREATE token  0 C, #256 CHARS ALLOT
```

```
: getname ( -- c-addr u )
        Look alpha? 0= IF  S" Name" expected  ENDIF
        token C0!
        BEGIN  Look alnum?
        WHILE  Look >UPC token char+!  getchar
        REPEAT token COUNT ;
-- --------------------------------------------------------------
-- Get a number

CREATE #value 0 C, #256 CHARS ALLOT

: getnum ( -- c-addr u )
        Look digit? 0= IF  S" Integer" expected  ENDIF
        #value C0!
        BEGIN   Look digit?
        WHILE   Look #value char+!  getchar
        REPEAT  #value COUNT ;
-- --------------------------------------------------------------
```

(Заметьте, что эта версия GetNum возвращает строку, а не целое число, как прежде).

Вы можете легко проверить что эти подпрограммы работают, вызвав их из основной программы :

```
 GetName TYPE
```

Эта программа выведет любое допустимое набранное имя (максимум восемь знаков, потому что мы так сказали GetName). Она отвергнет что-либо другое(chap7a.frt).

Аналогично проверьте другую подпрограмму.

ПРОБЕЛ

Раньше мы также работали с вложенными пробелами, используя две подпрограммы IsWhite и SkipWhite. Удостоверьтесь, что эти подпрограммы есть в вашей текущей версии Cradle и добавьте строку:

```
SkipWhite
```

в конец GetName и GetNum.

Теперь давайте определим новую процедуру:

```
-- --------------------------------------------------------------
-- Lexical scanner
: scan ( -- c-addr u )
   Look alpha? IF  getname EXIT  ENDIF
   Look digit? IF  getnum  EXIT  ENDIF
   Look token C! token 1  getchar skipwhite ;
-- --------------------------------------------------------------
```

Мы можем вызвать ее из новой основной программы:

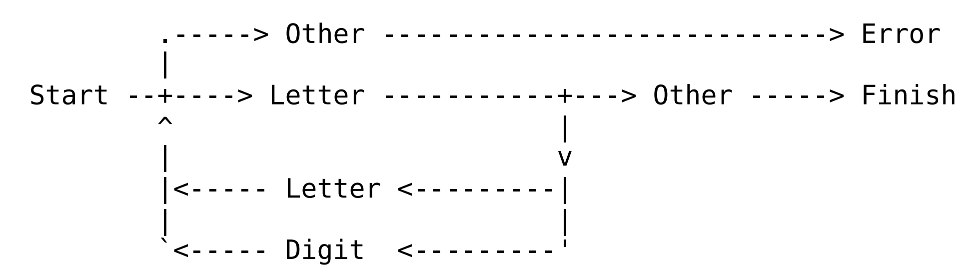
```
-- --------------------------------------------------------------
-- Main Program
   init
   BEGIN
      scan token$ PACK COUNT CR TYPE
      token$ CHAR+ C@ ^M =
   UNTIL ;
-- --------------------------------------------------------------
```

(Вы должны добавить описание строки Token в начало программы. Сделайте ее любой удобной длины, скажем 16 символов).

Теперь запустите программу.(chap7b.frt) Заметьте, что входная строка действительно разделяется на отдельные токены.

КОНЕЧНЫЕ АВТОМАТЫ

Подпрограмма анализа типа GetName действительно реализует конечный автомат. Состояние неявно в текущей позиции в коде. Очень полезным приемом для визуализации того, что происходит, является синтаксическая диаграмма или "railroad-track" диаграмма. Немного трудно нарисовать их в этой среде, поэтому я буду использовать их очень экономно, но фигура ниже должна дать вам идею:



Как вы можете видеть, эта диаграмма показывает логические потоки по мере чтения символов. Начинается все, конечно, с состояния "start" и заканчивается когда найден символ, отличный от алфавитно-цифрового. Если первый символ не буква, происходит ошибка. Иначе автомат продолжит выполнение цикла до тех пор, пока не будет найден конечный разделитель.

Заметьте, что в любой точке потока наша позиция полностью зависит от предыдущей истории входных символов. В этой точке предпринимаемые действия зависят только от текущего состояния плюс текущий входной символ. Это и есть то, что образует конечный автомат.

Из-за сложностей представления "railroad-track" диаграмм в этой среде я буду продолжать придерживаться с этого времени синтаксических уравнений. Но я настоятельно рекомендую вам диаграммы для всего, что включает синтаксический анализ. После небольшой практики вы можете начать видеть, как написать синтаксический анализатор непосредственно из диаграммы. Параллельные пути кодируются в контролирующие действия (с помощью операторов IF или CASE), последовательные пути - в последовательные вызовы. Это почти как работа по схеме.

Мы даже не обсудили SkipWhite, которая была представлена раньше, но это также простой конечный автомат, как и GetNum. Так же как и их родительская процедура Scan. Маленькие автоматы образуют большие автоматы.

Интересная вещь, на которую я хотел бы чтобы вы обратили внимание это то, как безболезненно такой неявный подход создает эти конечные автоматы. Я лично предпочитаю его таблично-управляемому методу. Он также получает маленькие, компактные и быстрые сканеры.

НОВЫЕ СТРОКИ

Продвигаясь прямо вперед, давайте модифицируем наш сканер для поддержки более чем одной строки. Как я упомянул последний раз, наиболее простой способ сделать это - просто обработать символы новой строки, возврат каретки и перевод строки, как незаполненное пространство. Фактически это способ, используемый подпрограммой iswhite из стандартной библиотеки C. Прежде мы не этого делали. Я хотел бы сделать это теперь, чтобы вы могли почувствовать результат.

Чтобы сделать это просто измените единственную выполнимую строку в IsWhite:

```
: white? ( char -- tf )
	DUP  Tab =
	OVER BL  = OR
```

```
OVER ^M  = OR
SWAP ^J  = OR ;
```

Мы должны дать основной программы новое условие останова, так как она никогда не увидит CR. Давайте просто используем:

```
token$ CHAR+ C@ '.' = UNTIL ;
```

OK, откомпилируйте эту программу(chap7c.frt) и запустите ее. Попробуйте пару строк, завершаемых точкой. Я использовал:

```
now is the time
for all good men.
```

Эй, что случилось? Когда я набрал это, я не получил последний токен, точку. Программа не остановилась. Более того, когда я нажал клавишу 'enter' несколько раз, я все равно не получил точку.

Если вы все еще не можете выбраться из вашей программы, вы обнаружите, что набор точки в новой строке прервет ее.

Что здесь происходит? Ответ в том, что мы зависаем в SkipWhite. Короткий осмотр этой подпрограммы покажет, что пока мы печатаем пустые строки, мы просто продолжаем выполнение цикла. После того, как SkipWhite встречает LF, он пытается выполнить GetChar. Но так как входной буфер теперь пуст, оператор чтения в GetChar настаивает на наличии другой строки. Процедура Scan получает завершающую точку, все правильно, но она вызывает SkipWhite и SkipWhite не возвращается до тех пор, пока не получит непустую строку.

Такое поведение не настолько плохое, как кажется. В настоящем компиляторе мы читали бы символы из входного файла вместо консоли и пока мы имеем какую-то процедуру для работы с концом файла, все получится OK. Но для чтения данных с консоли такое поведение слишком причудливое. Суть в том, что соглашение C/Unix просто не совместимо со структурой нашего анализатора, который запрашивает предсказывающий символ. Код, который мастера из Bell реализовали, не использует это соглашение, поэтому они нуждаются в 'ungetc'.

OK, давайте исправим проблему. Чтобы сделать это, мы должны возвратиться к старому определению IsWhite (удалите символы CR и LF) и используйте процедуру Fin, которую я представил в последний раз. Если ее нет в вашей текущей версии Cradle, поместите ее там.

Также измените основную программу следующим обраазом:

```
-- -------------------------------------------------------------
-- Main Program
: main ( -- )
   init
   BEGIN
      scan token$ PACK COUNT CR TYPE
      token$ CHAR+ C@ ^M  = IF  fin  ENDIF
      token$ CHAR+ C@ '.' =
   UNTIL ;
-- -------------------------------------------------------------
```

Обратите внимание на "охраняющую" проверку, предшествующую вызову Fin. Это то, что заставляет все это работать, и проверяет, то мы не пытаемся прочитать строку дальше.

Сейчас испытайте этот код. Я думаю он понравится вам больше.

Если вы обратитесь к коду, который мы написали в последней главе, вы обнаружите, что я расставил вызовы Fin по всему коду, где прерывание строки было бы уместным. Это одна из тех областей, которые действительно влияют на восприятие, о котором я упомянул. В этой точке я должен убедить вас поэкспериментировать с различными способами организациями

и посмотреть, как вам это понравится. Если вы хотите, чтобы ваш язык был по настоящему свободного стиля, тогда новые строки должны быть прозрачны. В этом случае наилучшим подходом было бы поместить следующие строки в начале Scan:

```
BEGIN  Look ^M =
WHILE  fin
REPEAT
```

Если, с другой стороны, вам нужен строчно-ориентированный язык подобный Ассемблеру, BASIC или FORTRAN (или даже Ada... заметьте, что он имеет комментарии, завершаемые новой строкой), тогда вам необходимо, чтобы Scan возвращал CR как токены. Он также должен съедать завершающие LF. Лучший способ сделать - использовать эту строку в самом начале Scan:

```
Look ^J = IF  fin  ENDIF
```

Для других соглашений вы будете должны использовать другие способы организации. В моем примере на последнем уроке я разрешил новые строки только в определенных местах, поэтому я занял какое-то промежуточное положение. В остальных частях этих занятий я буду выбирать такие способы обработки новых строк какие мне понравятся, но я хочу, чтобы вы знали, как выбрать для себя другой путь.

ОПЕРАТОРЫ

Мы могли бы сейчас остановиться и иметь в своем распоряжении довольно полезный сканер. В тех фрагментах KISS, которые мы построили, единственными токенами, состоящими из нескольких символов, являются идентификаторы и числа. Все операторы были односимвольными. Единственное исключение, которое я могу придумать - это операторы отношений "<=", ">=" и "<>", но они могут быть обработаны как особые случаи.

Однако другие языки имеют многосимвольные операторы такие как ":=" в Паскале или "++" и ">>" в C. Хотя пока нам и не нужны многосимвольные операторы, было бы хорошо знать как получить их в случае необходимости.

Само собой разумеется, что мы можем обрабатывать операторы точно таким же способом, что и другие токены. Давайте начнем с подпрограммы распознавания:

```
-- -------------------------------------------------------------
-- Recognize any operator
: op? ( char -- tf )
    DUP  '+' =
    OVER '-' = OR
    OVER '*' = OR
    OVER '/' = OR
    OVER '<' = OR
    OVER '>' = OR
    OVER ':' = OR
    SWAP '=' = OR ;
-- -------------------------------------------------------------
```

Важно заметить, что мы не должны включать в этот список каждый возможный оператор. К примеру круглые скобки не включены, так же как и завершающая точка. Текущая версия Scan и так хорошо поддерживает односимвольные операторы. Список выше включает только те символы, которые могут появиться в многосимвольных операторах. (Для конкретных языков список конечно всегда может быть отредактирован).

Теперь давайте изменим Scan следующим образом:

```
-- -------------------------------------------------------------
-- Get an operator
: getop ( -- )
   token$ C0!
```

```
   Look op? 0= IF  S" Operator" expected  ENDIF
   BEGIN Look op?
   WHILE Look token$ char+!  getchar
   REPEAT
   token$ COUNT  skipwhite ;
-- --------------------------------------------------------------
-- Lexical scanner
: scan ( -- c-addr u )
   BEGIN  Look ^M = WHILE  fin  REPEAT
   Look alpha? IF  getname EXIT  ENDIF
   Look digit? IF  getnum  EXIT  ENDIF
   Look op?    IF  getop   EXIT  ENDIF
   Look token$ char-place
   token$ COUNT  getchar skipwhite ;
-- --------------------------------------------------------------
```

Теперь испытайте программу.(chap7e.frt) Вы убедитесь, что любые фрагменты кода, которые вы захотите бросить в нее будут аккуратно разложены на индивидуальные токены.

СПИСКИ, ЗАПЯТЫЕ И КОМАНДНЫЕ СТРОКИ.

Прежде чем возвратиться к основной цели нашего обучения, я хотел бы немного выступить.

Сколько раз вы работали с программой или операционной системой, которая имела жесткие правила того, как вы должны разделять элементы в списке? (Попробую, последний раз вы использовали MS DOS!). Некоторые программы требуют пробелов как разделителей, некоторые требуют запятые. Хуже всего, что некоторые требуют и того и другого в разных местах. Большинство довольно неумолимы к нарушениям их правил.

Я думаю, это непростительно. Слишком просто написать синтаксически анализатор, который поддерживает и пробелы и запятые гибким способом. Рассмотрите следующую процедуру:

```
-- --------------------------------------------------------------
-- Skip over a comma
: skipcomma ( -- )
   skipwhite
   Look ',' = IF  getchar skipwhite  ENDIF ;
-- --------------------------------------------------------------
```

Эта процедура из восьми строк пропустит разделитель, состоящий из любого числа (включая ноль) пробелов, с нулем или одной запятой, вложенной в строку.

Временно измените вызов SkipWhite в Scan на вызов SkipComma и попробуйте ввести какие-нибудь списки. Хорошо работает, да? Разве вы не хотите, чтобы больше создателей программ знало о SkipComma?

К слову сказать, я обнаружил, что добавление эквивалента SkipComma в мою программу на ассемблере для Z80 заняло всего шесть дополнительных байт кода. Даже на 64K машинах это не слишком большая цена за дружелюбие к пользователю.

Я думаю вы можете видеть к чему я клоню. Даже если вы в своей жизни не написали ни одной строчки кода для компилятора, в каждой программе существуют места, где вы можете использовать понятие синтаксического анализа. Любая программа, которая обрабатывает командные строки, нуждается в нем. Фактически, если вы подумаете немного об этом, вы придете к заключению, что всякий раз, когда вы пишете программу, обрабатывающую ввод пользователя, вы определяете язык. Люди общаются с помощью языков и неявный синтаксис в вашей программе определяет этот язык. Настоящий вопрос: вы собираетесь определять его преднамеренно и явно, или просто позволите существовать независимо от того, как программа завершает синтаксический анализ?

Я утверждаю, что у вас будет лучший, более дружественный интерфейс если вы

потратите время на то, чтобы определить синтаксис явно. Запишите синтаксические уравнения или нарисуйте "railroad-track" диаграммы и закодируйте синтаксический анализатор используя методы, которые я показал вам здесь. Вы получите более хорошую программу и ее будет проще писать, в придачу.

СТАНОВИТСЯ ИНТЕРЕСНЕЙ

Хорошо, сейчас мы имеем довольно хороший лексический анализатор, который разбивает входной поток на лексемы. Мы могли бы использовать его как есть и иметь полезный компилятор. Но есть некоторые другие аспекты лексического анализа, которые мы должны охватить.

Особенно следует рассмотреть (вздрогните) эффективность. Помните, когда мы работали с односимвольными токенами, каждой проверкой было сравнение одного символа Look с байтовой константой. Мы также использовали в основном оператор Case.

С многосимвольными лексемами, возвращаемыми Scan, все эти проверки становятся сравнением строк. Гораздо медленнее. И не только медленнее но и неудобней, так как в Паскале не существут строкового эквивалента оператора Case. Особенно расточительным кажется проверять то что состоит из одного символа... "=", "+" и другие операторы... используя сравнение строк.

Сравнение строк не является невозможным. Рон Кейн использовал этот подход при написании Small C. Так как мы придерживаемся принципа KISS мы были бы оправданы согласившись с этим подходом. Но тогда я не смог бы рассказать вам об одном из ключевых методов, используемых в "настоящих" компиляторах.

Вы должны запомнить: лексический анализатор будет вызываться часто! Фактически один раз для каждой лексемы во всей исходной программе. Эксперименты показали, что средний компилятор тратит где-то от 20 до 40 процентов своего времени на подпрограммах лексического анализа. Если существовало когда-либо место, где эффективность заслуживает пристального рассмотрения, то это оно.

По этой причине большинство создателей компиляторов заставляют лексический анализатор выполнять немного больше работы, "токенизируя" входной поток. Идея состоит в том, чтобы сравнивать каждую лексему со списком допустимых ключевых слов и операторов и возвращать уникальный код для каждой распознанной. В случае обычного имени переменной или числа мы просто возвращаем код, который говорит, к какому типу лексем они относятся и сохраняем где-нибудь текущую строку.

Первое, что нам нужно - это способ идентификации ключевых слов. Мы всегда можем сделать это с помощью последовательных проверок IF, но несомненно было бы хорошо, если бы мы имели универсальную подпрограмму, которая могла бы сравнивать данную строку с таблицей ключевых слов. (Между прочим, позднее нам понадобится такая же подпрограмма для работы с таблицей идентификаторов). Это обычно выявляет проблему Паскаля, потому что стандартный Паскаль не имеет массивов переменной длины. Это настоящая головная боль - обьявлять различные подпрограммы поиска для каждой таблицы. Стандартный Паскаль также не позволяет инициализировать массивы, поэтому вам придется видеть код типа:

```
S" IF"    1 table PACK DROP
S" ELSE"  2 table PACK DROP
.
.
S" END"   n table PACK DROP
```

что может получиться довольно длинным если есть много ключевых слов.

К счастью Turbo Pascal 4.0 имеет расширения, которые устраняют обе эти проблемы. Массивы-константы могут быть обьявлены с использованием средства TP "типизированные константы" а переменные размерности могут быть поддержаны с помощью Си-подобных

расширений для указателей.

Сначала, измените ваши объявления подобным образом:

```
-- --------------------------------------------------------------
-- Type declarations
: $, ( c-addr u size -- )
        >S S MIN >R  R@ C, R@ 0 ?DO  C@+ C,  LOOP DROP
        S> R> - CHARS ALLOT ;

: SYMTAB ( size -- )
        CREATE  HERE >R
                0 ,  ( #items )
                DUP >S ,  ( itemsize )
                BEGIN  BL <WORD> DUP
                WHILE  S $, 1 R@ +!
                REPEAT 2DROP -R -S
        DOES>   CELL+ @+ 1+ ROT * + ( ix -- addr ) ;
-- --------------------------------------------------------------
```

(Размерность, использованная в SymTab не настоящая... память не распределяется непосредственно этим объявлением, а размерность должна быть только "достаточно большой")

Затем, сразу после этих объявлений, добавьте следующее:

```
-- --------------------------------------------------------------
-- Definition of keywords and token types
8 CHARS =: /symbol
/symbol SYMTAB KWlist IF ELSE ENDIF END
-- --------------------------------------------------------------
```

Затем, вставьте следующую новую функцию:

```
-- --------------------------------------------------------------
-- Table lookup
-- If the input string matches a table entry, return the entry
-- index.  If not, return a -1.
: lookup ( c-addr u 'table -- n2 )
        0 0 LOCALS| /symbol n table sz addr |
        table 2 CELLS - @+ TO n  @ TO /symbol
        0 n 1- DO
                  /symbol 1+ I * table + COUNT
                  addr sz COMPARE 0= IF  I UNLOOP EXIT  ENDIF
         -1 +LOOP
        -1 ;
-- --------------------------------------------------------------
```

Чтобы проверить ее chap7f.frtвы можете временно изменить основную программу следующим образом:

```
-- --------------------------------------------------------------
  S" ENDIF" 0 KWlist lookup .  ( 2 )
-- --------------------------------------------------------------
```

Обратите внимание как вызывается Lookup: функция Addr устанавливает указатель на KWList, который передается в Lookup.

OK, испытайте ее. Так как здесь мы пропускаем Scan, для получения соответствия вы должны набирать ключевые слова в верхнем регистре.

Теперь, когда мы можем распознавать ключевые слова, далее необходимо договориться о возвращаемых для них кодах.

Итак, какие кода мы должны возвращать? В действительности есть только два

приемлемых варианта. Это похоже на идеальное применения перечислимого типа Паскаля. К примеру, вы можете определить что-то типа

```
: SymType ( u -- ) KWlist CONSTANT ;

   0 SymType IfSym
   1 SymType ElseSym
   2 SymType EndifSym
   3 SymType EndSym

   -1 =: unknown
   -2 =: Ident
   -3 =: Number
   -4 =: Operator
```

и договориться возвращать переменную этого типа. Давайте попробуем это. Вставьте строку выше в описание типов.

Теперь добавьте два описания переменных:

```
    0 VALUE token                  -- current token
    CREATE $token  16 CHARS ALLOT  -- string token of look
```

Измените сканер так:

```
-- ------------------------------------------------------------
-- Lexical scanner
: scan ( -- c-addr u )
   BEGIN Look ^M = WHILE  fin  REPEAT
   Look alpha? IF  getname 2DUP 0 KWlist lookup
                   DUP 0>= IF  KWlist
                         ELSE  DROP Ident
                        ENDIF  TO token
                   EXIT
            ENDIF
   Look digit? IF  getnum Number   TO token EXIT  ENDIF
   Look op?    IF  getop  Operator TO token EXIT  ENDIF
   Look token$ char-place  Operator TO token
   token$ COUNT  getchar skipwhite ;
-- ------------------------------------------------------------
```

(Заметьте, что Scan сейчас стала процедурой а не функцией).

Наконец, измените основную программу:

```
-- ------------------------------------------------------------
-- Main Program
: main ( -- )
   init
   BEGIN
      scan
      CASE token
        Ident    OF  ." Ident "    ENDOF
        Number   OF  ." Number "   ENDOF
        Operator OF  ." Operator " ENDOF
        IfSym    OF  ." Keyword "  ENDOF
        ElseSym  OF  ." Keyword "  ENDOF
        EndifSym OF  ." Keyword "  ENDOF
        EndSym   OF  ." Keyword "  ENDOF
        unknown  OF  ." unknown "  ENDOF
      ENDCASE    TYPE CR
      token EndSym =
   UNTIL ;
-- ------------------------------------------------------------
```

Мы заменили строку Token, используемую раньше, на перечислимый тип. Scan возвращает тип в переменной Token и возвращает саму строку в новой переменной Value.

OK, откомпилируйте программу и погоняйте е(chap7g.frt)е. Если все работает, вы должны увидеть, что теперь мы распознаем ключевые слова.

Теперь у нас все работает правильно, и было легко сгенерировать это из того, что мы имели раньше. Однако, она все равно кажется мне немного "перегруженной". Мы можем ее немного упростить, позволив GetName, GetNum, GetOp и Scan работать с глобальными переменными Token и Value, вследствие этого удаляя их локальные копии. Кажется немного умней было бы переместить просмотр таблицы в GetName. Тогда новая форма для этих четырех процедур будет такой:

```
-- ---------------------------------------------------------------
-- Get an Identifier
: getname ( -- c-addr u )
        Look alpha? 0= IF  S" Name" expected  ENDIF
        token$ C0!
        BEGIN  Look alnum?
        WHILE  Look >UPC token$ char+!  getchar
        REPEAT token$ COUNT  skipwhite
        0 KWlist lookup DUP 0>= IF  KWlist
                              ELSE  DROP Ident
                             ENDIF  TO token
        token$ COUNT ;
-- ---------------------------------------------------------------
-- Get a number
: getnum ( -- c-addr u )
        Look digit? 0= IF  S" Integer" expected  ENDIF
        token$ C0!
        BEGIN   Look digit?
        WHILE   Look token$ char+!  getchar
        REPEAT  Number TO token
        token$ COUNT  skipwhite ;
-- ---------------------------------------------------------------
-- Get an operator
: op? ( char -- tf )
        DUP  '+' =
        OVER '-' = OR
        OVER '*' = OR
        OVER '/' = OR
        OVER '<' = OR
        OVER '>' = OR
        OVER ':' = OR
        SWAP '=' = OR ;

: getop ( -- c-addr u )
        token$ C0!
        Look op? 0= IF  S" Operator" expected  ENDIF
        BEGIN  Look op?
        WHILE  Look token$ char+!  getchar
        REPEAT Operator TO token
        token$ COUNT  skipwhite ;

-- ---------------------------------------------------------------
-- Lexical scanner
: scan ( -- c-addr u )
   BEGIN Look ^M = WHILE  fin  REPEAT
   Look alpha? IF  getname EXIT  ENDIF
   Look digit? IF  getnum  EXIT  ENDIF
   Look op?    IF  getop   EXIT  ENDIF
   Look token$ char-place  Operator TO token
   token$ COUNT  getchar skipwhite ;
-- ---------------------------------------------------------------
```

ВОЗВРАЩЕНИЕ СИМВОЛА

По существу, все сканер, которые я когда-либо видел и которые написаны на Паскале, использовали механизм перечислимых типов, который я только что описал. Это конечно работающий механизм, но он не кажется мне самым простым подходом.

Прежде всего, список возможных типов символов может получиться довольно длинным. Здесь я использовал только один символ "Operator" для обозначения всех операторов, но я видел другие проекты, в которых фактически возвращаются различные кода для каждого.

Существует, конечно, другой простой тип, который может быть возвращен как код: символ. Вместо возвращения значения "Operator" для знака "+", что неправильного в том, чтобы просто возвращать сам символ? Символ - такая же хорошая переменная для кодирования различных типов лексем, она легко может быть использована в операторах Case, и это гораздо проще набрать. Что может быть проще?

Кроме того, мы уже имели опыт с идеей кодировать ключевые слова как одиночные символы. Наши предыдущие программы уже написаны таким способом, так что использование этого метода минимизирует изменения того, что мы уже сделали.

Некоторые из вас могут почувствовать, что идея с возвращение символьных кодов слишком детская. Я должен допустить, что она становится немного неуклюжей для операторов типа "<=". Если вы хотите остаться с перечислимыми типами, хорошо. Для остальных я хотел бы показать как изменить то, что мы сделали выше, для поддержки такого подхода.

Во-первых, сейчас вы можете удалить объявление типа SymType... он нам больше не понадобится. И вы можете изменить тип Token в char.

Затем, чтобы заменить SymType, добавьте следующую константу:

```
CREATE KWcode ," xilee"
```

(Я буду кодировать все идентификаторы одиночным символом 'x').

Наконец измените Scan и его родственников следующим образом:

```
-- ---------------------------------------------------------------
-- Get an identifier
: getname ( -- c-addr u )
        Look alpha? 0= IF  S" Name" expected  ENDIF
        token$ C0!
        BEGIN  Look alnum?
        WHILE  Look >UPC token$ char+!  getchar
        REPEAT token$ COUNT  skipwhite
        0 KWlist lookup KWcode + C@ TO token
        token$ COUNT ;
-- ---------------------------------------------------------------
-- Get a number
: getnum ( -- c-addr u )
        Look digit? 0= IF  S" Integer" expected  ENDIF
        token$ C0!
        BEGIN   Look digit?
        WHILE   Look token$ char+!  getchar
        REPEAT  '#' TO token
        token$ COUNT  skipwhite ;
-- ---------------------------------------------------------------
-- Get an operator
: op? ( char -- tf )
        DUP  '+' =
        OVER '-' = OR
        OVER '*' = OR
        OVER '/' = OR
        OVER '<' = OR
        OVER '>' = OR
        OVER ':' = OR
```

```
        SWAP '=' = OR ;

: getop ( -- c-addr u )
        token$ C0!
        Look op? 0= IF  S" Operator" expected  ENDIF
        BEGIN  Look op?
        WHILE  Look token$ char+!  getchar
        REPEAT '?' TO token
        token$ C@ 1 = IF  token$ CHAR+ C@
                        ELSE '?'
                       ENDIF TO token
        token$ COUNT  skipwhite ;
-- --------------------------------------------------------------
-- Lexical Scanner
: scan ( -- c-addr u )
   BEGIN Look ^M = WHILE  fin  REPEAT
   Look alpha? IF  getname EXIT  ENDIF
   Look digit? IF  getnum  EXIT  ENDIF
   Look op?    IF  getop   EXIT  ENDIF
   Look char-place  '?' TO token
   token$ COUNT  getchar skipwhite ;
-- --------------------------------------------------------------
-- Main Program
: main ( -- )
   init
   BEGIN
      scan
      CASE token
        'x' OF  ." Ident "    ENDOF
        '#' OF  ." Number "   ENDOF
        'i' OF  ." Keyword "  ENDOF
        'l' OF  ." Keyword "  ENDOF
        'e' OF  ." Keyword "  ENDOF
                ." Operator "
      ENDCASE
      token$ .$
      token$ COUNT S" END" COMPARE 0=
   UNTIL ;
-- --------------------------------------------------------------
```

Эта программа должна работать также как и предыдущая версия. Небольшое различие в структуре, может быть, но она кажется мне более простой.

РАСПРЕДЕЛЕННЫЕ СКАНЕРЫ ПРОТИВ ЦЕНТРАЛИЗОВАННЫХ

Структура лексического анализатора, которую я только что вам показал, весьма стандартна и примерно 99% всех компиляторов используют что-то очень близкое к ней. Это, однако, не единственно возможная структура, или даже не всегда самая лучшая.

Проблема со стандартным подходом состоит в том, что сканер не имеет никаких сведений о контексте. Например, он не может различить оператор присваивания "=" и оператор отношения "=" (возможно именно поэтому и C и Паскаль используют для них различные строки). Все, что сканер может сделать, это передать оператор синтаксическому анализатору, который может точно сказать исходя из контекста, какой это оператор. Точно так же, ключевое слово "IF" не может быть посредине арифметического выражения, но если ему случится оказаться там, сканер не увидит в этом никакой проблемы и возвратит его синтаксическому анализатору, правильно закодировав как "IF".

С таким подходом, мы в действительности не используем всю информацию, имеющуюся в нашем распоряжении. В середине выражения, например, синтаксический анализатор "знает", что нет нужды искать ключевое слово, но он не имеет никакой возможности сказать это сканеру. Так что сканер продолжает делать это. Это, конечно, замедляет компиляцию.

В настоящих компиляторах проектировщики часто принимают меры для передачи

подробной информации между сканером и парсером, только чтобы избежать такого рода проблем. Но это может быть неуклюже и, конечно, уничтожит часть модульности в структуре компилятора.

Альтернативой является поиск какого-то способа для использования контекстной информации, которая исходит из знания того, где мы находимся в синтаксическом анализаторе. Это возвращает нас обратно к понятию распределенного сканера, в котором различные части сканера вызываются в зависимости от контекста.

В языке KISS, как и большинстве языков, ключевые слова появляются только в начале утверждения. В таких местах, как выражения они запрещены. Также, с одним небольшим исключением (многосимвольные операторы отношений), которое легко обрабатывается, все операторы односимвольны, что означает, что нам совсем не нужен GetOp.

Так что, оказывается, даже с многосимвольными токенами мы все еще можем всегда точно определить вид лексемы исходя из текущего предсказывающего символа, исключая самое начало утверждения.

Даже в этой точке, единственным видом лексемы, который мы можем принять, является идентификатор. Нам необходимо только определить, является ли этот идентификатор ключевым словом или левой частью оператора присваивания.

Тогда мы заканчиваем все еще нуждаясь только в GetName и GetNum, которые используются так же, как мы использовали их в ранних главах.

Сначала вам может показаться, что это шаг назад и довольно примитивный способ. Фактически же, это усовершенствование классического сканера, так как мы используем подпрограммы сканирования только там, где они действительно нужны. В тех местах, где ключевые слова не разрешены, мы не замедляем компиляцию, ища их.

ОБЪЕДИНЕНИЕ СКАНЕРА И ПАРСЕРА

Теперь, когда мы охватили всю теорию и общие аспекты лексического анализа, я наконец готов подкрепит свое заявление о том, что мы можем приспособить многосимвольные токены с минимальными изменениями в нашей предыдущей работе. Для краткости и простоты я ограничу сам себя подмножеством того, что мы сделали ранее: я разрешу только одну управляющую конструкцию (IF) и никаких булевых выражений. Этого достаточно для демонстрации синтаксического анализа и ключевых слов и выражений. Расширение до полного набора конструкций должно быть довольно очевидно из того, что мы уже сделали.

Все элементы программы для синтаксического анализа этого подмножества с использованием односимвольных токенов уже существуют в наших предыдущих программах. Я построил ее осторожно скопировав эти файлы, но я не посмею попробовать провести вас через этот процесс. Вместо этого, во избежание беспорядка, вся программа показана ниже(kiss00.frt): :

```
-- -----------------------------------------------------------------------
-- Variable declarations -------------------------------------------------

0 VALUE Look            -- lookahead character
0 VALUE Lcount          -- label counter
0 VALUE token           -- encoded value of scanned token

CREATE token$  0 C, #256 CHARS ALLOT -- token in unencoded string form

-- Tools -----------------------------------------------------------------

: getchar  ( -- )          EKEY TO Look ;  -- read new character from input stream
: error    ( c-addr u -- ) CR ^G EMIT ." Error: " TYPE ." ." ;  -- report an error
: aborts   ( c-addr u -- ) error ABORT ;                    -- report error and halt
: expected ( c-addr u -- ) S"  Expected" $+ aborts ;   -- report what was expected
: alpha?   ( char -- tf )  >UPC 'A' 'Z' 1+ WITHIN ; -- recognize an alpha character
: digit?   ( char -- tf )  '0' '9' 1+ WITHIN ;        -- recognize a decimal digit
: alnum?   ( char -- tf )  DUP alpha? SWAP digit? OR ;   -- recognize alphanumeric
```

```
: white?   ( char -- tf )  DUP Tab =  SWAP BL  =  OR ;   -- recognize white space
: addop?   ( char -- tf )  DUP '+' =  SWAP '-' =  OR ;          -- test for AddOp
: mulop?   ( char -- tf )  DUP '*' =  SWAP '/' =  OR ;          -- test for MulOp
: emits    ( c-addr u -- ) Tab EMIT TYPE ;            -- output a string with tab
: emitln   ( c-addr u -- ) CR emits ;        -- output a string with tab and crlf

-- skip cr+lf
: fin ( -- )
        Look ^M = IF getchar ENDIF
        Look ^J = IF getchar ENDIF ;

-- skip white space
: skipwhite ( -- )
        BEGIN  Look white?
        WHILE  getchar
        REPEAT ;

-- match a specific input character
: match ( char -- )
        DUP Look = IF  DROP getchar
                 ELSE  S" `" ROT CHAR-APPEND &' CHAR-APPEND expected
                ENDIF ;

-- get an identifier.
: getname ( -- char )
        BEGIN  Look ^M = WHILE  fin  REPEAT
        Look alpha? 0= IF  S" Name" expected  ENDIF
        Look >UPC  getchar skipwhite ; ( Returns a single character for now )

-- get a number
: getnum ( -- char )
        Look digit? 0= IF  S" Integer" expected  ENDIF
        Look  getchar skipwhite ; ( Returns a single character for now )

-- generate a unique label
: newlabel ( -- c-addr u ) S" @" Lcount U>D <# #S #> $+  1 +TO Lcount ;

-- post a label to output
: postlabel ( c-addr u -- ) CR TYPE ':' EMIT ;

-- parse and translate an identifier.
-- Note that we need to store getname locally because parsing a call's
-- parameterlist may clobber it.
: ident ( -- )
        getname LOCAL name
        Look '(' = IF   '(' match ')' match
                        S"  offset NEAR  call,"
                 ELSE   S"  dword-ptr -> eax  mov,"
                ENDIF   name CHAR-PREPEND emitln ;

DEFER expression

-- parse and translate a math factor
: factor ( -- )
        Look '('  = IF  '(' match  expression  ')' match EXIT  ENDIF
        Look alpha? IF  ident EXIT  ENDIF
        S"  d# -> eax  mov," getnum CHAR-PREPEND emitln ;

-- parse and translate the first math factor
: signedfactor ( -- )
        Look '+'  = IF  getchar       ENDIF
        Look '-' <> IF  factor  EXIT  ENDIF
        getchar
        Look digit? IF  S" #-" getnum CHAR-APPEND S"  d# -> eax  mov," $+
                  ELSE  factor S" eax  neg,"
                 ENDIF  emitln ;
```

```
-- recognize and translate multiply
: multiply ( -- )
        '*' match  factor
        S" [esp] dword  mul, [esp 4 +] -> esp  lea," emitln ;

-- recognize and translate divide
: divide ( -- )
        '/' match  factor
        S" ecx pop, ecx -> eax  xchg,  eax -> edx  mov," emitln
        S" #31 b# -> edx  sar,  ebx  idiv," emitln ;

-- parse and translate a math term
: term ( -- )
        signedfactor
        BEGIN Look mulop?
        WHILE   S" eax push," emitln
                CASE Look
                  '*' OF multiply  ENDOF
                  '/' OF divide    ENDOF
                         S" Mulop" expected
                ENDCASE
        REPEAT ;

-- recognize and translate add
: add ( -- )
        '+' match  term
        S" [esp] -> eax  add,  [esp 4 +] -> esp  lea," emitln ;

-- recognize and translate subtract
: subtract ( -- )
        '-' match  term
        S" [esp] -> eax  sub,  [esp 4 +] -> esp  lea,  eax neg," emitln ;

-- parse and translate an expression
:NONAME ( -- )
        term
        BEGIN   Look addop?
        WHILE   S" eax push," emitln
                CASE Look
                  '+' OF add      ENDOF
                  '-' OF subtract ENDOF
                         S" Addop" expected
                ENDCASE
        REPEAT ; IS expression

-- Parse and translate a boolean condition
-- This is a dummy.
: condition ( -- ) S" condition" emitln ;

DEFER block

-- Recognize and translate an IF construct
: doIF ( -- )
        0 0 LOCALS| L2 L1 |
        'i' match
        condition
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L1 PACK DROP
        L1 COUNT DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP
        L1 COUNT S"  offset NEAR  je," $+ emitln
        block
        Look 'l' = IF   'l' match
                        L2 FREE ?ALLOCATE
                        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP
                        L2 COUNT S"  offset NEAR  jmp," $+ emitln
                        L1 COUNT postlabel
                        block
                ENDIF
```

```
        'e' match
        L2 COUNT postlabel

        L1 FREE ?ALLOCATE
        L2 FREE ?ALLOCATE ;

-- Parse and translate an assignment statement
: assignment ( -- )
        getname LOCAL name
        '=' match
        expression
        S" eax -> " 'OF name 1 $+ S"  dword-ptr  mov," $+ emitln ;

-- Recognize and translate a statement block
:NONAME ( -- )
        BEGIN  Look 'e' <>
               Look 'l' <> AND
        WHILE  CASE Look
                'i' OF  doIF                              ENDOF
                ^M  OF  BEGIN Look ^M = WHILE fin REPEAT  ENDOF
                        assignment
               ENDCASE
        REPEAT ; IS block

-- Parse and translate a program
: doprogram ( -- )
        block
        Look 'e' <> IF  S" End" expected  ENDIF
        S" END" emitln ;

-- initialize
: init ( -- )
        CLEAR Lcount
        CR getchar skipwhite ;

: KISS00 ( -- ) init doprogram ;
-- -----------------------------------------------------------------------------
```

Пара комментариев:

- Форма синтаксического анализатора выражений, использующего FirstTerm и т.п., немного отличается от того, что вы видели ранее. Это еще одна вариация на ту же самую тему. Не позволяйте им вертеть вами... изменения необязательны для того, что будет дальше.
- Заметьте, что как обычно я добавил вызовы Fin в стратегических местах для поддержки множественных строк.

Прежде чем приступить к добавлению сканера, сначала скопируйте этот файл и проверьте, что он действительно корректно выполняет анализ. Не забудьте "кода": "i" для IF, "l" для ELSE и "e" для ELSE или ENDIF.

Если программа работает, тогда давайте поспешим. При добавлении модулей сканера в программу поможет систематический план. Во всех синтаксических анализаторах, которые мы написали до этого времени, мы придерживались соглашения, что текущий предсказывающий символ должен всегда быть непустым символом. Мы предварительно загружали предсказывающий символ в Init и после этого оставляли "помпу запущенной". Чтобы позволить программе работать правильно с новыми строками мы должны ее немного модифицировать и обрабатывать символ новой строки как допустимый токен.

В многосимвольной версии правило аналогично: текущий предсказыващий символ должен всегда оставаться на начале следующей лексемы или на новой строке.

Многосимвольная версия показана ниже. Чтобы получить ее я сделал следующие изменения:

- Добавлены переменные Token и Value и определения типов, необходимые для Lookup.
- Добавлено определение KWList и KWcode.
- Добавлен Lookup.
- GetName и GetNum заменены их многосимвольными версиями. (Обратите внимание, что вызов Lookup был перемещен из GetName, так что он не будет выполняться внутри выражений).
- Создана новая, рудиментарная Scan, которая вызывает GetName затем сканирует ключевые слова.
- Создана новая процедура MatchString, которая ищет конкретное ключевое слово. Заметьте, что в отличие от Match, MatchString не считывает следующее ключевое слово.
- Изменен Block для вызова Scan.
- Немного изменены вызовы Fin. Fin теперь вызывается из GetName.

Программа полностью:(kiss01.frt)

```
-- --------------------------------------------------------
-- Variable declarations ----------------------------

-1 =: unknown

0 VALUE Lcount                          -- label counter
0 VALUE Look                            -- lookahead character
0 VALUE token                           -- encoded token

CREATE token$  0 C, #16 CHARS ALLOT     -- unencoded token

-- helper to build symbol tables
: $, ( c-addr u size -- )
        >S S MIN >R
        R@ C, R@ 0 ?DO  C@+ C,  LOOP DROP
        S> R> - CHARS ALLOT ;

-- Type declarations
: SYMTAB ( size -- )
        CREATE  HERE >R
                0 ,  ( #items )
                DUP >S ,  ( itemsize )
                BEGIN  BL <WORD> DUP
                WHILE  S $, 1 R@ +!
                REPEAT 2DROP -R -S
        DOES>   CELL+ @+ 1+ ROT * + ( ix -- addr ) ;

-- Definition of keywords and token types

8 CHARS =: /symbol

/symbol SYMTAB KWlist IF ELSE ENDIF WHILE ENDWHILE READ WRITE VAR END

: KW->token ( kw -- ) 2+ C" xilewhRWve" + C@ TO token ;

: lookup ( c-addr u 'table -- n2|unknown )
        0 0 LOCALS| /symbol n table sz addr |
        table 2 CELLS - @+ TO n  @ TO /symbol
        0 n 1- DO
                  /symbol 1+ I * table + COUNT
                  addr sz COMPARE 0= IF  I UNLOOP EXIT  ENDIF
         -1 +LOOP
        unknown ;
```

```
-- Tools -----------------------------------------------------------------------

: getchar  ( -- )          EKEY TO Look ;                          -- read new char
: error    ( c-addr u -- ) CR ^G EMIT ." Error: " TYPE ." ." ;  -- report an error
: aborts   ( c-addr u -- ) error ABORT ;                           -- report and halt
: expected ( c-addr u -- ) S"  Expected" $+ aborts ;               -- report expected
: alpha?   ( char -- tf )  >UPC 'A' 'Z' 1+ WITHIN ;                -- is alpha char?
: digit?   ( char -- tf )  '0' '9' 1+ WITHIN ;                     -- is digit?
: alnum?   ( char -- tf )  DUP alpha? SWAP digit? OR ;             -- is alphanum?
: white?   ( char -- tf )  DUP Tab =  SWAP BL  =  OR ;             -- is white space?
: orop?    ( char -- tf )  DUP '|' =  SWAP '~' =  OR ;             -- recognize ORs
: addop?   ( char -- tf )  DUP '+' =  SWAP '-' =  OR ;             -- test for AddOps
: mulop?   ( char -- tf )  DUP '*' =  SWAP '/' =  OR ;             -- test for MulOps
: emits    ( c-addr u -- ) Tab EMIT TYPE ;                         -- output with tab
: emitln   ( c-addr u -- ) CR emits ;                              -- output with crlf

: char+!   ( c addr -- )   DUP >S COUNT + C!  1 S> C+! ;

-- skip white space
: skipwhite ( -- )
        BEGIN  Look white?
        WHILE  getchar
        REPEAT ;

-- skip cr+lf
: fin ( -- )
        Look ^M = IF getchar ENDIF
        Look ^J = IF getchar ENDIF
        skipwhite ;

-- match a specific input character
: match ( char -- )
        DUP Look = IF  DROP getchar skipwhite
                 ELSE  S" `" ROT CHAR-APPEND &' CHAR-APPEND expected
                ENDIF ;

-- get an identifier.
: getname ( -- c-addr u )
        BEGIN  Look ^M = WHILE  fin  REPEAT
        Look alpha? 0= IF  S" Name" expected  ENDIF
        token$ C0!
        BEGIN   Look alnum?
        WHILE   Look >UPC token$ char+!  getchar
        REPEAT
        token$ COUNT  skipwhite ;

-- get a number
: getnum ( -- c-addr u )
        Look digit? 0= IF  S" Integer" expected  ENDIF
        token$ C0!
        BEGIN   Look digit?
        WHILE   Look token$ char+!  getchar
        REPEAT  '#' TO token
        token$ COUNT  skipwhite ;

-- Get an identifier and scan it for keywords
: scan ( -- ) getname 0 KWlist lookup KW->token ;

-- Match a specific input string
: matchstring ( c-addr u -- )
        token$ COUNT 2OVER
        COMPARE IF  &` CHAR-PREPEND &' CHAR-APPEND expected
              ELSE  2DROP
             ENDIF ;

-- generate a unique label
```

```
: newlabel ( -- c-addr u ) S" @" Lcount U>D <# #S #> $+  1 +TO Lcount ;

-- post a label to output
: postlabel ( c-addr u -- ) CR TYPE ':' EMIT ;

-- parse and translate an identifier.
-- Note that we need to store getname locally because parsing a call's
-- parameterlist may clobber it.
: ident ( -- )
        getname DUP 1+ ALLOCATE ?ALLOCATE DUP LOCAL name PACK DROP
        Look '(' = IF   '(' match ')' match
                        name COUNT S"  offset NEAR  call,"
                 ELSE   name COUNT S"  dword-ptr -> eax  mov,"
                ENDIF   $+ emitln
        name FREE ?ALLOCATE ;

DEFER expression

-- parse and translate a math factor
: factor ( -- )
        Look '('  = IF  '(' match  expression  ')' match EXIT  ENDIF
        Look alpha? IF  ident EXIT  ENDIF
        getnum S"  d# -> eax  mov," $+ emitln ;

-- parse and translate the first math factor
: signedfactor ( -- )
        Look '+'  = IF  getchar       ENDIF
        Look '-' <> IF  factor  EXIT  ENDIF
        getchar skipwhite
        Look digit? IF  S" #-" getnum $+ S"  d# -> eax  mov," $+
                  ELSE  factor S" eax  neg,"
                 ENDIF  emitln ;

-- recognize and translate multiply
: multiply  ( -- )
        '*' match  factor
        S" [esp] dword  mul, [esp 4 +] -> esp  lea," emitln ;

-- recognize and translate divide
: divide ( -- )
        '/' match  factor
        S" ecx pop, ecx -> eax  xchg,  eax -> edx  mov," emitln
        S" #31 b# -> edx  sar,  ebx  idiv," emitln ;

-- parse and translate a math term
: term ( -- )
        signedfactor
        BEGIN Look mulop?
        WHILE   S" eax push," emitln
                CASE Look
                  '*' OF multiply  ENDOF
                  '/' OF divide    ENDOF
                         S" Mulop" expected
                ENDCASE
        REPEAT ;

-- recognize and translate add
: add ( -- )
        '+' match  term
        S" [esp] -> eax  add,  [esp 4 +] -> esp  lea," emitln ;

-- recognize and translate subtract
: subtract ( -- )
        '-' match  term
        S" [esp] -> eax  sub,  [esp 4 +] -> esp  lea,  eax neg," emitln ;

-- parse and translate an expression
```

```
:NONAME ( -- )
        term
        BEGIN   Look addop?
        WHILE   S" eax push," emitln
                CASE Look
                  '+' OF add      ENDOF
                  '-' OF subtract ENDOF
                         S" Addop" expected
                ENDCASE
        REPEAT ; IS expression

-- Parse and translate a boolean condition
-- This is a dummy.
: condition ( -- ) S" condition" emitln ;

DEFER block

-- Recognize and translate an IF construct
: doIF ( -- )
        0 0 LOCALS| L2 L1 |
        condition
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L1 PACK DROP
        L1 COUNT DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP
        L1 COUNT S"  offset NEAR  je," $+ emitln
        block
        token 'l' = IF  L2 FREE ?ALLOCATE
                        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP
                        L2 COUNT S"  offset NEAR  jmp," $+ emitln
                        L1 COUNT postlabel
                        block
                 ENDIF
        L2 COUNT postlabel
        S" ENDIF" matchstring

        L1 FREE ?ALLOCATE
        L2 FREE ?ALLOCATE ;

-- Parse and translate an assignment statement
: assignment ( -- )
        token$ COUNT DUP 1+ ALLOCATE ?ALLOCATE DUP LOCAL name PACK DROP
        '=' match
        expression
        S" eax -> " name COUNT $+ S"  dword-ptr  mov," $+ emitln
        name FREE ?ALLOCATE ;

-- Recognize and translate a statement block
:NONAME ( -- )
        scan
        BEGIN  token 'e' <>
               token 'l' <> AND
        WHILE  CASE token
                'i' OF  doIF  ENDOF
                        assignment
               ENDCASE
               scan
        REPEAT ; IS block

-- Parse and translate a program
: doprogram ( -- )
        block
        token 'e' <> IF  S" End" expected  ENDIF
        S" END" emitln ;

-- initialize
: init ( -- )
        CLEAR Lcount
        CR getchar skipwhite ;
```

```
: KISS01 ( -- ) init doprogram ;
-- ---------------------------------------------------------------------
```

Сравните эту программу с ее односимвольным вариантом. Я думаю вы согласитесь, что различия минимальны.

ЗАКЛЮЧЕНИЕ

К этому времени вы узнали как анализировать и генерировать код для выражений, булевых выражений и управляющих структур. Теперь вы изучили, как разрабатывать лексические анализаторы и как встроить их элементы в транслятор. Вы все еще не видели всех элементов, объединеных в одну программу, но на основе того, что мы сделали ранее вы должны прийти к заключению, что легко расширить наши ранние программы для включения лексических анализаторов.

Мы очень близки к получению всех элементов, необходимых для построения настоящего, функционального компилятора. Есть еще несколько отсутствующих вещей, особенно вызовы процедур и определения типов. Мы будем работать с ними на следующих нескольких уроках. Прежде чем сделать это, однако, я подумал что было бы забавно превратить транслятор в настоящий компилятор. Это то, чем мы займемся в следующей главе.

До настоящего времени мы применяли предпочтительно восходящий метод синтаксического анализа, начиная с низкоуровневых конструкций и продвигаясь вверх. В следующей главе я также взгляну сверху вниз, и мы обсудим, как изменяется структура транслятора при изменении определения языка.

Увидимся.

8. Немного философии

ВВЕДЕНИЕ

Этот урок будет отличаться от других уроков в нашей серии по синтаксическому анализу и конструированию компиляторов. На этом уроке не будет никаких экспериментов или кода. На этот раз я хотел бы просто поговорить с вами некоторое время. К счастью, это будет короткий урок и затем мы сможем продолжить с того места где остановились, надо надеяться с обновленной энергией.

Когда я учился в университете, я обнаружил, что могу всегда следить за профессорской лекцией намного лучше, если знал куда он идет. Готов поспорить, с вами было то же самое.

Так что я подумал, может быть пришло время расказать вам куда мы идем с этой серией: что нас ждет в будущих главах и вообще что к чему. Я также поделюсь своими общими мыслями о полезности того, что мы делали.

ДОРОГА ДОМОЙ

Пока что мы охватили синтаксический анализ и трансляцию арифметических выражений, булевых выражений и их комбинаций, связанных операторами отношений. Мы также сделали то же самое для управляющих конструкций. Во всем этом мы склонялись в основном к использованию нисходящего синтаксического анализа методом рекурсивного спуска, определение синтаксиса в БНФ и непосредственной генерации ассемблерного кода. Мы также изучили значение такого приема как односимвольные токены. В последней главе мы работали с лексическим анализом и я показал вам простой но мощный способ преодоления односимвольного барьера.

В течение всех этих исследований, я особенно выделял философию KISS… Keep It Simple, Sidney... и я надеюсь, что к настоящему времени вы поняли, насколько простыми могут в действительности быть эти вещи. Хотя наверняка имеются области в теории компиляции которые являются по настоящему пугающими, основной мыслью этой серии является то, что

на практике вы можете просто вежливо обойти многие из этих областей. Если определение языка способствует этому или, как в этой серии, если вы можете определить язык по ходу дела, то возможно записать определение языка в БНФ с достаточным удобством. И, как мы видели, вы можете вывести процедуры синтаксического анализа из БНФ почти также быстро, как вы можете набирать на клавиатуре.

По мере того, как наш компилятор принимал некоторую форму, он приобретал больше частей, но каждая часть довольно мала и проста и очень похожа на все другие.

К этому моменту у нас есть многое из того, что составляет настоящий практический компилятор. Фактически, мы уже имеем все что нам нужно для создания игрушечного языка столь же мощного, как, скажем, Tiny Basic. В следующих двух главах мы пойдем вперед и определим этот язык.

Для завершения этой серии, у нас все еще есть несколько тем для раскрытия. Они включают:

- Вызовы процедур, с параметрами и без.
- Локальные и глобальные переменные.
- Базовые типы, такие как символьные и целочисленные типы.
- Массивы.
- Строки.
- Типы и структуры, определяемые пользователем.
- Синтаксические анализаторы с деревьями и промежуточные языки.
- Оптимизация.

Все это будет рассмотрено в будущих главах. Когда мы закончим, вы будете иметь все инструменты, необходимые для разработки и создания своего собственного языка и компиляторов для его трансляции.

Я не могу спроектировать эти языки для вас, но я могу дать некоторые комментарии и рекомендации. Я уже высказал некоторые из них в прошлых главах. Вы видели, например, какие управляющие структуры я предпочитаю.

Эти конструкции будут частью создаваемых мной языков. К этому моменту я представляю три языка, два из которых вы увидите в очередных главах:

TINY - минимальный, но пригодный для использования язык уровня Tiny Basic или Tiny C. Он не будет очень практичным, но будет достаточно мощным, чтобы позволить вам писать и запускать настоящие программы которые делают что-нибудь заслуживающее внимание.

KISS - язык, который я создаю для своего собственного использования. KISS предназначен быть языком системного программирования. Он не будет иметь строгого контроля типов или причудливых структур данных, но он будет поддерживать большинство вещей, которые я хочу делать с языком более высокого уровня (HOL), за исключением возможно написания компиляторов.

Я также играл в течение нескольких лет с идеей HOL-подобного ассемблера со структурными управляющими конструкциями и HOL-подобными операциями присваивания. Это фактически было стимулом для моего первоначального углубления в джунгли теории компиляции. Этот язык возможно никогда не будет создан просто потому, что я узнал, что проще реализовать язык типа KISS, который использует только подмножество инструкций ЦПУ. Как вы знаете, ассемблер может быть предельно причудливым и нерегулярным, и язык, который отображается в него один к одному, может быть настоящим вызовом. Однако я всегда чувствовал, что синтаксис, используемый в стандартных ассемблерах тупой… почему

```
ebx -> eax mov,
```

лучше или проще для трансляции, чем

```
B=A ?
```

Я думаю, было бы интересным упражнением разработка "компилятора" который дал бы программисту полный доступ и к контролю над полным набором инструкций ЦПУ, и позволил бы вам генерировать программы настолько же эффективные как язык ассемблер без болезненного изучения набора мнемоник. Это может быть сделано? Я не знаю. Настоящим вопросом может быть вопрос "будет ли полученный язык проще, чем ассемблер?" Если нет, то в нем нет никакого смысла. Я думаю, что это может быть сделано, но я полностью еще не уверен в том, как должен выглядеть синтаксис.

Возможно у вас есть некоторые комментарии или предложения об этом. Буду рад услышать их.

Вы возможно не будете удивлены узнав, что уже работал в большинстве тех областей, которые мы рассмотрим. Я имею несколько хороших новостей: дела никогда не будут намного более сложными, чем они были до этого. Возможно построить завершенный, работающий компилятор для реального языка используя только те самые методы которые вы изучили до этого. И это поднимет некоторые интересные вопросы.

ПОЧЕМУ ЭТО ТАК ПРОСТО?

Перед осуществлением этой серии я всегда думал, что компиляторы были просто естественно сложными компьютерными программами... предельно вызывающими. Однако то, что мы здесь делали обычно оказывалось совершенно простым, иногда даже тривиальным.

Некоторое время я думал, что это было просто потому, что я еще не залез в глубь темы. Я только охватил простые части. Я легко признаюсь вам что даже когда я начинал эту серию я не был уверен в том, как далеко мы будем способны продвинуться прежде чем дела станут слишком сложными для работы имеющимися способами. Но сейчас я уже нахожусь достаточно близко, чтобы увидеть конец пути. Какой вывод?

ЗДЕСЬ НЕТ НИЧЕГО СЛОЖНОГО!

Затем я думал, что причина в том, что мы не генерировали очень хороший объектный код. Те из вас, кто следовали этой серии и пытались компилировать примеры, знают, что хотя код работает и достаточно отказоустойчив, его эффективность довольно ужасна. Я подчеркивал, что если бы мы сконцентрировались на получении компактного кода, то быстро бы получили всю недостающую сложность.

В какой то степени это так. В частности, мои первые небольшие усилия при попытке повысить эффективность подняли сложность до опасного уровня. Но с той поры когда я возился с некоторыми простыми методами оптимизацией и обнаружил некоторые, которые приводят к очень приличному качеству кода без добавления больших сложностей.

Наконец я подумал, что возможно причина была в "игрушечной" природе компилятора Я не претендовал на то, что мы когда-нибудь будем способны построить компилятор, конкурирующий с Borland и Microsoft. И однако снова, когда я забираюсь глубже в эти дела различия начинают стираться.

Просто чтобы удостовериться что до вас дошла эта мысль, позвольте мне ее высказать напрямую:

Используя методы, которые мы здесь применяли, возможно создать работающий, промышленного качества компилятор не добавляя много сложности к тому, что мы уже сделали.

С тех пор, как началась эта серия, я получил от вас некоторые комментарии. Большинство из них повторяют мои собственные мысли: "Это просто! Почему учебники представляют это настолько сложным?" Хороший вопрос.

Недавно я возвратился и взглянул на некоторые из этих текстов снова и даже купил и читаю некоторые новые. Каждый раз я возвращался с тем же чувством: эти ребята представляют это слишком сложным.

Что происходит? Почему все это кажется сложным в этих книгах, но легким для нас? Действительно ли мы умней чем Ахо, Ульман, Бринч Хансен и все остальные?

Едва ли. Но мы делаем некоторые вещи по-другому и все более и более я начинаю ценить значение нашего подхода и способ, которым он упрощает дело. Кроме очевидных сокращений, которые я выделил в первой части, типа односимвольных токенов и консольного ввода/вывода, мы сделали некоторые неявные предположения и сделали некоторые вещи отличными от того, как разрабатывали компиляторы в прошлом. Как оказалось, наш метод делает жизнь намного проще.

Но почемы все другие ребята не используют его?

Вы должны вспомнить контекст некоторых ранних разработок компиляторов. Эти люди работали на очень небольших компьютерах с ограниченными возможностями. Объемы памяти были очень ограничены, набор команд центрального процессора был минимален и программы чаще выполнялись в пакетном режиме, чем в интерактивном. Как оказалось, это повлияло на некоторые ключевые решения проекта, которые действительно усложнили проект. До недавнего времени я не понимал, насколько классический дизайн компилятора был обусловлен доступным оборудованием.

Даже в тех случаях, где эти ограничения больше не накладывались, люди предпочитали структурировать их программы тем же самым образом, так как это способ, которому они обучались.

В нашем случае мы начали с чистого листа бумаги. Имеется опасность, конечно, что вы попадетесь в ловушки, которые другие люди давно научились избегать. Но это также позволило нам использовать различные подходы, которые, частично из-за проекта, частично из-за чистой удачи, позволили нам добиться простоты.

Имеются области, которые, я думаю, в прошлом приводили к сложности:

- Ограниченная оперативная память, вынуждающая выполнять множество проходов.

Я только что прочитал "Brinch Hansen on Pascal Compilers" (отличная книга, BTW). Он разработал компилятор Pascal для PC, но он начал в 1981 г. с систем с 64K памяти и поэтому почти каждое решение проекта который он делал, было нацелено на то, чтобы уместить компилятор в ОЗУ. Чтобы сделать это, его компилятор выполнял три прохода, один из которых - лексический анализ. Не было никакого способа, с помощью которого он мог бы, например, использовать распределенный сканер, который я представил в последней главе, потому что структура программы не позволяла этого. Ему также требовались не один а два промежуточных языка для обеспечения связи между фазами.

Все ранние создатели компиляторов были вынуждены иметь дело с такой проблемой: разбить компилятор на достаточные части так, чтобы они поместились в памяти. Когда у вас есть множество проходов, вы должны добавить структуры данных для поддержки информации которую каждый проход оставляет для следующего. Это добавляет сложность и завершает управление проектом. В книге Ли "The Anatomy of a Compiler" упоминается компилятор Fortran, разработанный для IBM 1401. Он имел не менее 63 отдельных проходов! Само собой разумеется, в компиляторе, подобном этому, разделение на фазы доминировало бы над дизайном.

Даже в ситуации, когда ОЗУ достаточно, люди предпочитали использовать те же самые методы, с которыми они хорошо знакомы. До тех пор, пока не появился Turbo Pascal, мы не

знали насколько может быть простым компилятор если бы вы начали с других предположений.

- Пакетная обработка.

В ранние дни пакетная обработка была единственным выбором... не существовало никаких интерактивных вычислений. Даже сегодня компиляторы по существу выполняются в пакетном режиме.

В компиляторах для майнфреймов, так же как и во многих микро компиляторах, значительные усилия расходуются на восстановление после ошибок... это может занять 30-40% компилятора и полностью управлять проектом. Идея состоит в том, чтобы избежать остановки на первой ошибке, а скорее идти любой ценой, так чтобы вы могли сказать программисту о как можно большем количестве ошибок во всей программе насколько возможно.

Все это возвращает нас к дням ранних майнфреймов, где время выполнения измерялось в часах и днях и было важно выжать каждую последнюю унцию информации из каждого выполнения.

В этой серии я был очень осторожен и избежал проблемы восстановления после ошибок и вместо этого наш компилятор просто останавливается с сообщением на первой ошибке. Я откровенно признаюсь, что это было в основном потому, что я захотел использовать легкий путь и сохранить простоту. Но этот метод, заданный изначально Borland в Turbo Pascal также имеет много полезного в любом случае. Кроме сохранения простоты компилятора это также очень хорошо соответствует идее интерактивной системы. Когда компиляция происходит быстро и, особенно, когда вы имеете редактор типа Borland который будет правильно указывать вам на точку ошибки, тогда имеет смысл остановиться там и просто перезапустить компиляцию после того, как ошибка исправлена.

- Большие программы.

Ранние компиляторы были разработаны для поддержки больших программ... по существу бесконечных. В те дни существовал небольшой выбор; идея с библиотеками подпрограмм и раздельной компиляцией была еще в будущем. Снова, это предположение вело к многопроходному дизайну и промежуточным файлам для поддержки результатов раздельной обработки.

Поставленная Бринч Хансеном цель состояла в том, чтобы компилятор был способен компилировать сам себя. Снова, из-за ограничений оперативной памяти это приводило его к многопроходному дизайну. Он нуждался в таком маленьком резидентном коде компилятора, насколько возможно, так чтобы необходимые таблицы и другие структуры данных поместились в оперативную память.

Я не заявил об этом пока, потому что не было необходимости... мы всегда просто читали и записывали данные как потоки, в любом случае. Но, для заметки, мой план всегда был в том, чтобы в промышленном компиляторе исходные и обьектные данные должны сосуществовать в ОЗУ с компилятором, аля ранний Turbo Pascal. Вот почему я был осторожен и сохранил подпрограммы типа GetChar и Emit как отдельные попрограммы, несмотря на их небольшой размер. Будет просто изменить их на чтение и запись из памяти.

- Акцент на эффективность.

Джон Бэкус заявил, что когда он и его коллеги разработали первоначальный компилятор Fortran они знали, что они должны получать компактный код. В те дни имелись сильные чувства против HOL в пользу ассемблера и причиной была эффективность. Если бы Fortran не производил очень хороший код по стандартам ассемблера, пользователи просто бы отказались использовать его. Заметьте, компилятор Fortran оказался одним из наиболее эффективных из когда либо созданных.в терминах качества кода. Но он был сложным!

Сегодня мы имеем мощъ ЦПУ и размер ОЗУ с запасом, так что эффективность кода не такая большая проблема. Старательно игнорируя эту проблему мы действительно были способны сохранить простоту. Как ни странно, тем не менее, как я сказал, я нашел некоторую оптимизацию которую мы можем добавить в базовую структуру компилятора не добавляя слишком много сложности. Так что в этом случае мы получим свой пирог и сьедим его: мы в любом случае закончим с приемлемым качеством кода.

- Ограниченный набор инструкций.

Первые компьютеры имели примитивный набор команд. Вещи, которые мы считаем само собой разумеющимися такие как операции со стеком и косвенная адресация появились с большими сложностями.

Пример: в большинстве компиляторов имеется структура данных, называемая литерный пул (literal pool). Компилятор обычно идентифицирует все литералы, используемые в программе и собирает их в одиночную структуру данных. Все ссылки на литералы сделаны косвенно на этот пул. В конце компиляции компилятор выдает команды для выделения памяти и инициализации литерного пула.

Нам пока не нужно было обращаться к этой проблеме. Когда нам нужно загрузить литерал мы просто делаем это строкой:

```
 3 b# -> eax mov,
```

Можно кое-что упомянуть об использования литерного пула особенно на машине типа 8086, где данные и код могут быть разделены. Однако все это добавляет довольно большое количество сложности с небольшим результатом.

Конечно, без стека мы бы потерялись. И вызовы подпрограмм и временная память сильно зависят от стека и мы использовали его даже больше, чем необходимо для облегчения синтаксического анализа выражений.

- Желание общности.

Многое из содержимого типичной книги по компиляторам акцентировано на вопросы, к которым мы совсем не обращались... вопросы типа автоматической трансляции грамматик или генерация таблиц LALR анализа. Это не просто, потому что авторы хотят впечатлить вас. Имеются хорошие практические причины, почему эти темы рассмотрены здесь.

Мы концентрировались на использовании синтаксического анализатора с рекурсивным спуском для анализа детерминированной грамматики, т.е. грамматики, которая однозначна и, следовательно, может быть проанализирована с одним уровнем предсказывания. Я не сделал из этого большого ограничения, но факт то, что это представляет небольшое подмножество возможных грамматик. Фактически, существует бесконечное число грамматик, которые мы не можем анализировать используя наш метод. LR метод более мощный и может работать с теми грамматиками, с которыми мы не можем.

В теории компиляции важно знать, как работать с этими другими грамматиками и как преобразовать их в грамматики которые проще для работы с ними. К примеру многие (но не все) неоднозначные грамматики могут быть преобразованы в однозначные. Способ сделать это не всегда очевиден, всетаки, и так много людей посвятили годы на разработку способа их автоматического преобразования.

На практике, эти проблемы оказываются значительно менее важными. Современные языки стараются разрабатывать так, чтобы они были простыми для анализа в любом случае. Это было ключевой мотивацией при разработке Pascal. Несомненно, имеются паталогические грамматики, для которых вы с большим трудом написали бы однозначную БНФ, но в реальном мире лучшим ответом возможно было бы избежание этих грамматик.

В нашем случае, конечно, мы трусливо позволили языку развиваться по ходу дела. Вы не можете всегда иметь такую роскошь. Однако, с небольшой заботой вы были бы способны

сохранить синтаксический анализатор простым без необходимости прибегать к автоматическому переводу грамматик.

В этой серии мы приняли значительно отличающийся подход. Мы начали с чистого листа бумаги и разработали методы, которые работают в том контексте, в котором мы находимся: это однопользовательский персональный компьютер с вполне достаточно мощным ЦПУ и объемом ОЗУ. Мы ограничили сами себя приемлемыми грамматиками, которые легки для анализа, мы с успехом использовали систему команд ЦПУ, и мы не концентрировались на эффективности. Именно поэтому это было просто.

Означает ли это, что мы навсегда обречены создавать только игрушечные компиляторы? Нет, я так не думаю. Я уже сказал, что мы можем добавить некоторую оптимизацию без изменения структуры компилятора. Если мы захотим обрабатывать большие файлы, мы всегда можем добавить для этого буферизацию файлов. Эти вещи не оказывают влияния на общий дизайн компилятора.

И я думаю что это главный фактор. Начав с маленьких и ограниченных случаев мы были способны сконцентрироваться на структуре компилятора, которая естественна для работы. Так как структура естественным образом удовлетворяет работе, она почти обречена быть простой и прозрачной. Добавление возможностей не должно изменять основную структуру. Мы можем просто добавить расширения типа файловой структуры или добавить уровень оптимизации. Я считаю, что когда ресурсы были ограничены, структуры, которые люди получали, были искуственно искажены чтобы заставить их работать в этих условиях, и не были оптимальными структурами для имеющейся проблемы.

ЗАКЛЮЧЕНИЕ

В любом случае, это мое личное предположение каким образом у нас была возможность сохранить простоту. Мы начали с чего-то простого и позволили ему развиваться естественным образом, не пытаясь направит его в какое-то традиционное русло.

Мы собираемся продолжать таким же образом. Я дал вам список областей, которые мы охватим в следующих главах. После прочтения этих глав вы будете способны создавать законченные, работающие компиляторы почти для любого случая и делать это легко. Если вы действительно хотите создать компилятор промышленного качества вы сможете сделать и это также.

Для тех из вас, кто застоялся в ожидании кода для синтаксического анализатора, я приношу извинения за это отклонение. Я просто подумал, что вы хотели бы немного рассмотреть дела в перспективе. В следующий раз мы вернемся к основной цели обучения.

Пока что мы рассмотрели только части компиляторов и хотя мы имеем многое из завершенного языка мы не говорили о том как сложить все это вместе. Это будет темой наших следующих двух глав. Затем мы поспешим к новым темам, которые я указал в начале этой главы.

Увидимся.

9. Вид сверху

ВВЕДЕНИЕ

В предыдущих главах мы изучили многие из методов, необходимых для создания полноценного компилятора. Мы разработали операции присваивания (с булевыми и арифметическими выражениями), операторы отношений и управляющие конструкции. Мы все еще не обращались к вопросу вызова процедур и функций, но даже без них мы могли бы в принципе создать мини-язык. Я всегда думал, что было бы забавно просто посмотреть, насколько маленьким можно было бы построить язык, чтобы он все еще оставался полезным. Теперь мы уже почти готовы сделать это. Существует проблема: хотя мы знаем, как анализировать и транслировать конструкции, мы все еще совершенно не знаем, как сложить

их все вместе в язык.

В этих ранних главах разработка наших программ имела явно восходящий характер. В случае с синтаксическим анализом выражений, например, мы начали с самых низкоуровневых конструкций, индивидуальных констант и переменных и прошли свой путь до более сложных выражений.

Большинство людей считают, что нисходящий способ разработки лучше, чем восходящий. Я тоже так думаю, но способ, который мы использовали, казался естественно достаточным для тех вещей, которые мы анализировали.

Тем не менее вы не должны думать, что последовательный подход, который мы применяли во всех этих главах, является принципиально восходящим. В этой главе я хотел бы показать вам, что этот подход может работать точно также, когда применяется сверху вниз... может быть даже лучше. Мы рассмотрим языки типа C и Pascal и увидим как могут быть построены законченные компиляторы начиная сверху.

В следующей главе мы применим ту же самую методику для создания законченного транслятора подмножества языка KISS, который я буду называть TINY. Но одна из моих целей в этой серии состоит в том, чтобы вы не только могли увидеть как работает компилятор для TINY или KISS, но чтобы вы также могли разрабатывать и создавать компиляторы своих собственных языков. Примеры Си и Паскаля помогут вам в этом. Одна вещь, которую я хотел чтобы вы увидели, состоит в том, что естественная структура компилятора очень сильно зависит от транслируемого языка, поэтому простота и легкость конструирования компилятора очень сильно зависит от того, позволите ли вы языку определять структуру программы.

Немного сложнее получить полный компилятор C или Pascal, да мы и не будем. Но мы можем расчистить верхние уровни так, чтобы вы увидели как это делается.

Давайте начнем.

ВЕРХНИЙ УРОВЕНЬ

Одна из самых больших ошибок людей при нисходящем проектировании заключается в неправильном выборе истинной вершины. Они думают, что знают какой должна быть общая структура проекта и поэтому они продолжают и записывают ее.

Всякий раз, когда я начинаю новый проект, я всегда хочу сделать это в самом начале. На языке разработки программ (program design language - PDL) этот верхний уровень походит на что-нибудь вроде:

```
begin
   solve the problem
end
```

Конечно, я соглашусь с вами, что это не слишком большая подсказка о том, что расположено на следующем уровене, но я все равно запишу это просто для того, чтобы почувствовать, что я действительно начинаю с вершины.

В нашем случае, общая функция компилятора заключается в компиляции законченной программы. С этого начинается любое определение языка, записанное в БНФ. На что походит верхний уровень БНФ? Хорошо, это немного зависит от транслируемого языка. Давайте взглянем на Pascal.

СТРУКТУРА ПАСКАЛЯ

Большинство книг по Pascal включают БНФ определение языка. Вот несколько первых строк одного из них:

```
<program> ::= <program-header> <block> '.'

<program-header> ::= PROGRAM <ident>
```

```
<block> ::= <declarations> <statements>
```

Мы можем написать подпрограммы распознавания для работы с каждым из этих элементов подобно тому, как мы делали это прежде. Для каждого из них мы будем использовать знакомые нам односимвольные токены, затем понемногу расширяя их. Давайте начнем с первого распознавателя: непосредственно программы.

Для ее трансляции мы начнем с новой копии Cradle. Так как мы возвращаемся к односимвольным именам мы будем просто использовать "p" для обозначения "program".

К новой копии Cradle добавьте следующий код и вставьте обращение к нему из основной программы:

```
-- ---------------------------------------------------------------
-- Parse and translate a program
: prog ( -- )
        0 LOCAL name
        'p' match       \ Handles program header part
        getname TO name
        name prolog
        '.' match
        name epilog ;
-- ---------------------------------------------------------------
```

Процедуры Prolog и Epilog выполняют все, что необходимо для связи программы с операционной системой так чтобы она могла выполняться как программа. Само собой разумеется, эта часть будет очень ОС-зависима. Помните, что я выдаю код для 68000, работающий под ОС, которую я использую - SK*DOS. Я понимаю, что большинство из вас использует PC и вы предпочли бы увидеть что-нибудь другое, но я слишком далеко зашел, чтобы что-то сейчас менять!

В любом случае, SK*DOS особенно простая для общения операционная система. Вот код для Prolog и Epilog:

```
-- ---------------------------------------------------------------
-- Write the prolog
: prolog ( char -- )
        S" CODE " ROT CHAR-APPEND emitln
        S" rpush," emitln ;
-- ---------------------------------------------------------------
-- Write the epilog
: epilog ( char -- )
        S" rpop,  ebx jmp," emitln
        DROP S" END-CODE"   emitln ;
-- ---------------------------------------------------------------
```

Как обычно добавьте этот код и испытайте "компилятор".(chap9a.frt) В настоящее время существует только одна допустимая входная последовательность:

```
px. (где x - это любая одиночная буква, имя программы).
```

Хорошо, как обычно наша первая попытка не очень впечатляет, но я уверен к настоящему времени вы знаете, что дальше станет интересней. Есть одна важная вещь, которую следует отметить: на выходе получается работающая, законченная и выполнимая программа (по крайней мере после того, как она будет ассемблирована).

Это очень важно. Приятная особенность нисходящего метода состоит в том, что на любом этапе вы можете компилировать подмножество завершенного языка и получить программу, которая будет работать на конечной машине. Отсюда, затем, нам необходимо только добавлять возможности, расширяя конструкции языка. Это очень похоже на то, что мы уже делали, за исключением того, что мы подходили к этому с другого конца.

РАСШИРЕНИЕ

Чтобы расширить компилятор мы должны просто работать с возможностями языка последовательно. Я хочу начать с пустой процедуры, которая ничего не делает, затем добавлять детали в пошаговом режиме. Давайте начнем с обработки блока в соответствии с его PDL выше. Мы можем сделать это в два этапа. Сначала добавьте пустую процедуру:

```
-- ---------------------------------------------------------------
-- Parse and translate a pascal block
: doblock ( char -- ) prolog ;
-- ---------------------------------------------------------------
```

и измените Prog следующим образом:

```
-- ---------------------------------------------------------------
-- Parse and translate a program
: prog ( char -- )
        0 LOCAL name
        'p' match      \ Handles program header part
        getname TO name
        name doblock
        '.' match
        name epilog ;
-- ---------------------------------------------------------------
```

Это конечно не должно изменить поведения программы, и не меняет. Но сейчас определение Prog закончено и мы можем перейти к расширению DoBlock. Это получается прямо из его БНФ определения:

```
-- ---------------------------------------------------------------
-- Parse and translate a pascal block
: doblock ( char -- )
        LOCAL name
        declarations
        name prolog
        'OF name 1 prolog
        name postlabel
        statements ;
-- ---------------------------------------------------------------
```

Процедура PostLabel была определена в главе по ветвлениям. Скопируйте ее в вашу копию Cradle.

Я возможно должен объяснить причину вставки метки. Это имеет отношение к работе SK*DOS. В отличие от некоторых других ОС, SK*DOS позволяет точке входа в основную программу находиться в любом месте программы. Все, что вы должны сделать, это дать этой точке имя. Вызов PostLabel помещает это имя как раз перед первым выполнимым утверждением в основной программе. Как SK*DOS узнает какая из множества меток является точкой входа, спросите вы? Та, которая соответствует утверждению END в конце программы.

Теперь нам нужны заглушки для процедур Declarations и Statements. Сделайте их пустыми процедурами как мы делали это раньше.

Программа в(chap9b.frt)се еще делает то же самое? Тогда мы можем перейти к следующему этапу.

ОБЪЯВЛЕНИЯ

БНФ для объявлений в Pascal такая:

```
<declarations> ::= ( <label list>    |
                     <constant list> |
```

```
                          <type list>     |
                          <variable list> |
                          <procedure>     |
                          <function>         )*
```

(Заметьте, что я использую более либеральное определение, используемое в Turbo Pascal. В определении стандартного Pascal каждая из этих частей должна следовать в определенном порядке относительно других).

Как обычно давайте позволим одиночным символам представлять каждый из этих типов объявлений. Новая форма для Declarations:

```
-- ---------------------------------------------------------------
-- Parse and translate the declaration part
: declarations ( -- )
        BEGIN
          Look 'l' =
          Look 'c' = OR
          Look 't' = OR
          Look 'v' = OR
          Look 'p' = OR
          Look 'f' = OR
        WHILE   CASE Look
                 'l' OF  labels      ENDOF
                 'c' OF  constants   ENDOF
                 't' OF  types       ENDOF
                 'v' OF  variables   ENDOF
                 'p' OF  doprocedure ENDOF
                 'f' OF  dofunction  ENDOF
                ENDCASE
        REPEAT ;
-- ---------------------------------------------------------------
```

Конечно, нам нужны процедуры-заглушки для каждого из этих типов объявлений. На этот раз они не могут быть совсем пустыми процедурами, так как иначе мы останемся с бесконечным циклом While. По крайней мере каждая подпрограмма распознавания должна съедать символ, который вызывает ее. Вставьте следующие процедуры:

```
-- ---------------------------------------------------------------
-- Process label statement
: labels ( -- ) 'l' match ;
-- ---------------------------------------------------------------
-- Process const statement
: constants ( -- ) 'c' match ;
-- ---------------------------------------------------------------
-- Process type statement
: types ( -- ) 't' match ;
-- ---------------------------------------------------------------
-- Process var statement
: variables ( -- ) 'v' match ;
-- ---------------------------------------------------------------
-- Process procedure definition
: doprocedure ( -- ) 'p' match ;
-- ---------------------------------------------------------------
-- Process function definition
: dofunction ( -- ) 'f' match ;
-- ---------------------------------------------------------------
```

Теперь испытайте компилятор(chap9c.frt) используя несколько характерных входных последовательностей. Вы можете смешивать объявления любым образом, каким вам нравится пока последним символом в программе не будет ".", указывающий на конец программы. Конечно, ни одно из этих объявлений фактически ничего не объявляет, так что вам не нужны

(и вы не можете использовать) любые символы, кроме тех, которые обозначают ключевые слова.

Мы можем расширить раздел операторов аналогичным способом. БНФ для него будет:

```
<statements> ::= <compound statement>

<compound statement> ::= BEGIN <statement> (';' <statement>) END
```

Заметьте, что утверждение может начинаться с любого идентификатора, исключая END. Так что первая пустой формой процедуры Statements будет:

```
-- -------------------------------------------------------------
-- Parse and translate the statement part
: statements ( -- )
        'b' match
        BEGIN  Look 'e' <>
        WHILE  getchar
        REPEAT
        'e' match ;
-- -------------------------------------------------------------
```

Сейчас компилятор (chap9d.frtпримет любое число объявлений, сопровождаемое блоком BEGIN основной программы. Сам этот блок может содержать любые символы (за исключением END), но они должны присутствовать.

Простейшая входная форма сейчас

'pxbe.'

Испытайте ее. Также попробуйте некоторые ее комбинации. Сделайте некоторые преднамеренные ошибки и посмотрите что произойдет.

К этому моменту вы должны начать видеть основную линию. Мы начинаем с пустого транслятора для обработки программы, затем в свою очередь мы расширяем каждую процедуру, основанную на ее БНФ определении. Подобно тому, как более низкоуровневые БНФ определения добавляют детали и развивают определения более высокого уровня, более низкоуровневые распознаватели будут анализировать более детально входную программу. Когда последняя заглушка будет расширена, компилятор будет закончен. Это нисходящая разработка/реализация в ее чистейшей форме.

Вы могли бы заметить, что даже хотя мы и добавляли процедуры, выходной результат программы не изменялся. Так и должно быть. На этих верхних уровнях не требуется никакой выдачи кода. Распознаватели функционируют просто как распознаватели. Они принимают входные последовательности, отлавливают плохие и направляют хорошие в нужные места, так что они делают свою работу. Если бы мы занимались этим немного дольше, код начал бы появляться.

Следующим шагом в нашем расширении должна возможно быть процедура Statements. Определение Pascal:

```
<statement> ::= <simple statement> | <structured statement>

<simple statement> ::= <assignment> | <procedure call> | null

<structured statement> ::= <compound statement> |
                           <if statement>       |
                           <case statement>     |
                           <while statement>    |
                           <repeat statement>   |
                           <for statement>      |
                           <with statement>
```

Это начинает выглядеть знакомыми. Фактически вы уже прошли через процесс синтаксического анализа и генерации кода и для операций присваивания и для управляющих структур. Это место, где верхний уровень встречается с нашим восходящим методом из предыдущих уроков. Конструкции будут немного отличаться от тех, которые мы использовали для KISS, но в этих различиях нет ничего, чего бы вы не смогли сделать.

Я думаю теперь вы можете получить представление об этом процессе. Мы начали с завершенного БНФ описания языка. Начиная с верхнего уровня мы закодировали распознаватели для этих БНФ утверждений используя процедуры-заглушки для распознавателей следующего уровня. Затем мы расширили более низкоуровневые утверждения один за другим.

Как оказывается, определение Pascal очень совместимо с использованием БНФ и БНФ описания этого языка существуют в избытке. Вооружившись таким описанием вы обнаружите, что довольно просто продолжить процесс, который мы начали.

Вы могли бы продолжить расширение этих конструкций, просто чтобы прочувствовать это. Я не ожидаю, что вы сможет завершить сейчас компилятор Паскаля... есть слишком много вещей таких как процедуры и типы, к которым мы еще не обращались... но могло бы быть полезным попробовать некоторые из более знакомых вещей. Вам было бы полезно увидеть выполнимые программы, появляющиеся с другого конца.

Если бы я собирался обратиться к вопросам которые мы еще не охватили, я предпочел бы сделать это в контексте KISS. Мы не пытаемся построить полный копилятор Pascal, поэтому я собираюсь остановить на этом расширение Pascal. Давайте взглянем на очень отличающийся язык.

СТРУКТУРА СИ

Язык C совсем другой вопрос, как вы увидите. Книги по C редко включают БНФ определения языка. Возможно дело в том, что этот язык очень сложен для описания в БНФ.

Одна из причин что я показываю вам сейчас эти структуры в том что я могу впечатлить вас двумя фактами:

1. Определение языка управляет структурой компилятора. Что работает для одного языка может быть бедствием для другого. Это очень плохая идея попытаться принудительно встроить данную структуру в компилятор. Скорее вы должны позволить БНФ управлять структурой, как мы делали здесь.
2. Язык, для которого сложно написать БНФ также будет возможно сложен для написания компилятора. Си - популярный язык и он имеет репутацию как позволяющий сделать практически все, что возможно. Несмотря на успех Small C, C является непростым для анализа языком.

Программа на C имеет меньше структур, чем ее аналог на Pascal. На верхнем уровне все в C является статическим объявлением или данных или функций. Мы можем зафиксировать эту мысль так:

```
<program> ::= ( <global declaration> )*

<global declaration> ::= <data declaration>  | <function>
```

В Small C функции могут иметь только тип по умолчанию int, который не объявлен. Это делает входную программу легкой для синтаксического анализа: первым токеном является или "int", "char" или имя функции. В Small C команды препроцессора также обрабатываются компилятором соответствующе, так что синтаксис становится:

```
<global declaration> ::= '#' <preprocessor command>  |
                         'int' <data list>           |
```

```
                                    'char' <data list>          |
                                    <ident> <function body>     |
```

Хотя мы в действительности больше заинтересованы здесь в полном С , я покажу вам код, соответствующий структуре верхнего уровня Small C.

```
-- ------------------------------------------------------------
-- Parse and translate a program
: prog ( -- )
        BEGIN   Look ^Z <>
        WHILE   CASE Look
                  '#' OF preproc  ENDOF
                  'i' OF intdecl  ENDOF
                  'c' OF chardecl ENDOF
                      int dofunction
                ENDCASE
        REPEAT ;
-- ------------------------------------------------------------
```

Обратите внимание, что я должен был использовать ^Z чтобы указать на конец исходного кода. C не имеет ключевого слова типа END или "." для индикации конца программы.

С полным Си все не так просто. Проблема возникает потому, что в полном Си функции могут также иметь типы. Так что когда компилятор видит ключевое слово типа "int" он все еще не знает ожидать ли объявления данных или определение функции. Дела становятся более сложными так как следующим токеном может быть не имя... он может начинаться с "*" или "(" или комбинаций этих двух.

Точнее говоря, БНФ для полного Си начинается с:

```
<program> ::= ( <top-level decl> )*

<top-level decl> ::= <function def> | <data decl>

<data decl> ::= [<class>] <type> <decl-list>

<function def> ::= [<class>] [<type>] <function decl>
```

Теперь вы можете увидеть проблему: первые две части обьявлений для данных и функций могут быть одинаковыми. Из-за неоднозначности в этой грамматике выше, она является неподходящей для рекурсивного синтаксического анализатора. Можем ли мы преобразовать ее в одну из подходящих? Да, с небольшой работой. Предположим мы запишем ее таким образом:

```
<top-level decl> ::= [<class>] <decl>

<decl> ::= <type> <typed decl> | <function decl>

<typed decl> ::= <data list> | <function decl>
```

Мы можем написать подпрограмму синтаксичесого анализа для определений классов и типов и позволять им отложить их сведения и продолжать выполнение даже не зная обрабатывается ли функция или объявление данных.

Для начала, наберите следующую версию основной программы:

```
-- ------------------------------------------------------------
-- Main Program
: main ( -- )
        init
        BEGIN
          Look ^Z <>
        WHILE
          getclass
```

```
        gettype
        topdecl
      REPEAT ;
-- --------------------------------------------------------------
```

На первый раз просто сделайте три процедуры-заглушки которые ничего не делают, а только вызывают GetChar.

Работает ли эта программа? Хорошо, было бы трудно не сделать это, так как мы в действительности не требовали от нее какой-либо работы. Уже говорилось, что компилятор Си примет практически все без отказа. Несомненно это правда для этого компилятора, потому что в действительности все, что он делает, это съедает входные символы до тех пор, пока не найдет ^Z.

Затем давайте заставим GetClass делать что-нибудь стоящее. Объявите глобальную переменную

```
0 VALUE class
```

и измените GetClass

```
-- --------------------------------------------------------------
-- Get a storage class specifier
: getclass ( -- )
        Look 'a' =  Look 'x' = OR  Look 's' = OR
           IF Look TO class
              getchar
         ELSE 'a' TO class
        ENDIF ;
-- --------------------------------------------------------------
```

Здесь я использовал три одиночных символа для представления трех классов памяти "auto", "extern" и "static". Это не единственные три возможных класса… есть также "register" и "typedef", но это должно дать вам представление. Заметьте, что класс по умолчанию "auto".

Мы можем сделать подобную вещь для типов. Введите следующую процедуру:

```
-- --------------------------------------------------------------
-- Get a type specifier
: gettype ( -- )
        BL TO typ
        Look 'u'
         = IF   'u' TO sign
                'i' TO typ
                getchar
         ELSE   's' TO sign
        ENDIF
        Look 'i' = Look 'l' = OR  Look 'c' = OR
           IF   Look TO typ
                getchar
        ENDIF ;
-- --------------------------------------------------------------
```

Обратите внимание, что вы должны добавить еще две глобальные переменные Sign и Typ.

С этими двумя процедурами компилятор будет обрабатывать определение классов и типов и сохранять их результаты. Мы можем сейчас обрабатывать остальные объявления.

Мы еще ни коим образом не выбрались из леса, потому что все еще существуют много сложностей только в определении типов до того, как мы дойдем даже до фактических данных или имен функций. Давайте притворимся на мгновение, что мы прошли все эти заслоны и следующим во входном потоке является имя. Если имя сопровождается левой скобкой, то мы имеем объявление функции. Если нет, то мы имеем по крайней мере один элемент данных, и

возможно список, каждый элемент которого может иметь инициализатор.

Вставьте следующую версию TopDecl:

```
-- -----------------------------------------------------------
-- Process a top-level declaration
: topdecl ( -- )
        getname LOCAL Name
        Look '(' = IF  name dofunc
                 ELSE  name dodata
                ENDIF ;
-- -----------------------------------------------------------
```

(Заметьте, что так как мы уже прочитали имя, мы должны передать его соответствующей подпрограмме.)

Наконец, добавьте две процедуры DoFunc и DoData:

```
-- ------------------------------------------------------------
-- Process a function definition
: dofunc ( char -- )
        LOCAL n
        '(' match
        ')' match
        '{' match
        '}' match
        typ BL = IF  'i' TO typ  ENDIF
        CR class EMIT SPACE sign EMIT SPACE typ EMIT
           ."  function " n EMIT ;
-- ------------------------------------------------------------
-- Process a Data Declaration
: dodata ( char -- )
        LOCAL n
        typ BL = IF  S" Type declaration" expected  ENDIF
        CR class EMIT SPACE sign EMIT SPACE typ EMIT
           ."  data " n EMIT
        BEGIN  Look ',' =
        WHILE  ',' match
               getname TO n
               CR class EMIT SPACE sign EMIT SPACE typ EMIT
                  ."  data " n EMIT
        REPEAT ';' match ;
-- -----------------------------------------------------------
```

Так как мы еще далеки от получения выполнимого кода, я решил чтобы эти две подпрограммы только сообщали нам, что они нашли.

Протестируйте эту программу(chap9e.frt). Для объявления данных дайте список, разделенный запятыми. Мы не можем пока еще обрабатывать инициализаторы. Мы также не можем обрабатывать списки параметров функций но символы "(){}" должны быть.

Мы все еще очень далеко от того, чтобы иметь компилятор C, но то что у нас есть обрабатывает правильные виды входных данных и распознает и хорошие и плохие входных данные. В процессе этого естественная структура компилятора начинает принимать форму.

Можем ли мы продолжать пока не получим что-то, что действует более похоже на компилятор. Конечно мы можем. Должны ли мы? Это другой вопрос. Я не знаю как вы, но у меня начинает кружиться голова, а мы все еще далеки от того, чтобы даже получить что-то кроме объявления данных.

К этому моменту, я думаю, вы можете видеть как структура компилятора развивается из определения языка. Структуры, которые мы увидели для наших двух примеров, Pascal и C, отличаются как день и ночь. Pascal был разработан, по крайней мере частично, чтобы быть легким для синтаксического анализа и это отразилось в компиляторе. Вообще, Pascal более структурирован и мы имеем более конкретные идеи какие виды конструкций ожидать в любой точке. В C наооборот, программа по существу является списком объявлений

завершаемых только концом файла.

Мы могли бы развивать обе эти структуры намного дальше, но помните, что наша цель здесь не в том, чтобы построить компилятор C или Pascal, а скорее изучать компиляторы вообще. Для тех из вас, кто хотят иметь дело с Pascal или C, я надеюсь, что дал вам достаточно начал чтобы вы могли взять их отсюда (хотя вам скоро понадобятся некоторые вещи, которые мы еще не охватили здесь, такие как типы и вызовы процедур). Остальные будьте со мной в следующей главе. Там я проведу вас через разработку законченного компилятора для TINY, подмножества KISS.

Увидимся.

10. Представление "TINY"

ВВЕДЕНИЕ

В последней главе я показал вам основную идею нисходящей разработки компилятора. Я показал вам первые несколько шагов этого процесса для компиляторов Pascal и C, но я остановился далеко от его завершения. Причина была проста: если мы собираемся построить настоящий, функциональный компилятор для какого-нибудь языка, я предпочел бы сделать это для KISS, языка, который я определил в этой обучающей серии.

В этой главе мы собираемся сделать это же для подмножества KISS, которое я решил назвать TINY.

Этот процесс по существу будет аналогичен выделенному в главе 9, за исключением одного заметного различия. В той главе я предложил вам начать с полного БНФ описания языка. Это было бы прекрасно для какого-нибудь языка типа Pascal или C, определения которого устоялись. В случае же с TINY, однако, мы еще не имеем полного описания... мы будем определять язык по ходу дела. Это нормально. Фактически, это предпочительней, так как мы можем немного подстраивать язык по ходу дела для сохранения простоты анализа.

Так что в последующей разработке мы фактически будем выполнять нисходящую разработку и языка и его компилятора. БНФ описание будет расти вместе с компилятором.

В ходе этого будет принят ряд решений, каждое из которых будет влиять на БНФ и, следовательно, характер языка. В каждой решающей точке я попытаюсь не забывать объяснять решение и разумное обоснование своего выбора. Если вам случится придерживаться другого мнения и вы предпочтете другой вариант, вы можете пойти своим путем. Сейчас вы имеет базу для этого. Я полагаю важно отметить, что ничего из того, что мы здесь делаем не подчинено каким-либо жесткими правилами. Когда вы разрабатываете свой язык вы не должны стесняться делать это своим способом.

Многие из вас могут сейчас спросить: зачем нужно начинать с самого начала? У нас есть работающее подмножество KISS как результат главы 7 (лексический анализ). Почему бы просто не раширить его как нужно? Ответ тройной. Прежде всего, я сделал несколько изменений для упрощения программы... типа изоляции процедур генерации кода, в результате чего мы можем более легко выполнять преобразование для различных машин. Во-вторых, я хочу, чтобы вы увидели что разработка действительно может быть выполнена сверху вниз как это подчеркнуто в последней главе. Наконец, нам всем нужна практика. Каждый раз, когда я прохожу через эти упражнения, я начинаю понимать немного больше, и вы будете тоже.

ПОДГОТОВКА

Много лет назад существовали языки, называемые Tiny BASIC, Tiny Pascal и Tiny C, каждый из которых был подмножеством своего полного родительского языка. Tiny BASIC, к примеру, имел только односимвольные имена переменных и глобальные переменные. Он поддерживал только один тип данных. Звучит знакомо? К этому моменту мы имеем почти все инструменты, необходимые для создания компилятора подобного этому.

Однако язык, называемый Tiny-такой-то все же несет некоторый багаж, унаследованный от своего родительского языка. Я часто задавался вопросом, хорошая ли это идея. Согласен, язык, основанный на каком-то родительском языке, будет иметь преимущество знакомости, но может также существовать некоторый особенный синтаксис, перенесенный из родительского языка, который может приводить к появлению ненужной сложности в компиляторе. (Нигде это не является большей истиной, чем в Small C).

Я задавался вопросом, насколько маленьким и простым может быть создан компилятор и при этом все еще быть полезным, если он разрабатывался из условия быть легким и для использования и для синтаксического анализа. Давайте выясним. Этот язык будет называться просто "TINY". Он является подмножеством KISS, который я также еще полностью не определил, что по крайней мере делает нас последовательными (!). Я полагаю вы могли бы назвать его TINY KISS. Но это открывает целую кучу проблем, так что давайте просто придерживаться имени TINY.

Главные ограничения TINY будут возникать из-за тех вещей, которые мы еще не рассмотрели, таких как типы данных. Подобно своим кузенам Tiny C и Tiny BASIC, TINY будет иметь только один тип данных, 16-разрядное целое число. Первая версия, которую мы разработаем, не будет также иметь вызовов процедур и будет использовать односимвольные имена переменных, хотя, как вы увидите, мы можем удалить эти ограничения без особых усилий.

Язык, который я придумал, разделит некоторые хорошие особенности Pascal, C и Ada. Получив урок из сравнения компиляторов Pascal и C в предыдущей главе, TINY все же будет иметь преимущественно вкус Паскаля. Везде, где возможно, структура языка будет ограничена ключевыми словами или символами, так что синтаксический анализатор будет знать, что происходит без догадок.

Другое основное правило: Я хотел бы чтобы в течение всей разработки компилятор производил настоящий выполнимый код. Даже если его не может быть слишком много в самом начале, но по крайней мере он должен быть корректным.

Наконец, я буду использовать пару ограничений Pascal, которые имеют смысл: Все данные и процедуры должны быть объявлены перед тем, как они используются. Это имеет большой смысл, даже если сейчас единственным типом данных, который мы будем использовать, будет слово. Это правило, в свою очередь, означает, что единственное приемлемое место для размещения выполнимого кода основной программы - в конце листинга.

Определение верхнего уровня будет аналогично Pascal:

```
<program> ::= PROGRAM <top-level decl> <main> '.'
```

Мы уже достигли решающей точки. Моей первой мыслью было сделать основной блок необязательным. Кажется бессмысленным писать "программу" без основной программы, но это имеет смысл, если мы разрешим множественные модули, связанные вместе. Фактически я предполагаю учесть это в KISS. Но тогда мы столкнемся с кучей проблем, которые я предпочел бы сейчас не затрагивать. Например, термин "PROGRAM" в действительности становится неправильно употребляемым. MODULE из Modula-2 или UNIT из Turbo Pascal были бы более подходящими. Во-вторых, как насчет правил видимости? Нам необходимо соглашение для работы с видимостью имен в модулях. На данный момент лучше просто сохранить простоту и совершенно игнорировать эту идею.

Также необходимо определиться с требованием, чтобы основная программа была последней. Я играл с идеей сделать ее размещение нефиксированным как в C. Характер SK*DOS, ОС под которую я компилирую, позволяет сделать это очень просто. Но это в действительности не имеет большого смысла принимая во внимание Pascal-подобное требование, что все данные и процедуры должны быть обьявлены прежде чем они используются. Так как основная программа может вызывать только те процедуры, которые уже были объявлены, единственное местоположение, имеющее смысл - в конце, a la Pascal.

По данной выше БНФ давайте напишем синтаксический анализатор, который просто распознает скобки:

```
-- -------------------------------------------------------------
-- Parse and translate a program
: prog ( -- )
        'p' match
        header
        prolog
        '.' match
        epilog ;
-- -------------------------------------------------------------
```

Процедура Header просто выдает инициализационный код, необходимый ассемблеру:

```
-- -------------------------------------------------------------
-- Write header info
: header ( -- )
        S" -- DATA section --------" emitdln
        S" " emitdln ;
-- -------------------------------------------------------------
```

Процедуры Prolog и Epilog выдают код для идентификации основной программы и для возвращения в ОС:

```
-- -------------------------------------------------------------
-- Write the prolog
: prolog ( -- )
        S" " emitln
        S" -- CODE section --------" emitdln
        S" "          emitdln
        S" CODE main" emitdln
        S" rpush,"    emitln
        newlabel postlabel ;
-- -------------------------------------------------------------
-- Write the epilog
: epilog ( -- )
        S" rpop,  ebx jmp," emitln
        S" END-CODE"        emitdln ;
-- -------------------------------------------------------------
```

Основная программа просто вызывает Prog и затем выполняет проверку на чистое завершение:

```
-- --------------------------------------------------------------------------
: tiny10 ( -- )
        init prog
        Look ^M <> IF  S" Unexpected data after " S" ." name.aborts  ENDIF ;
-- --------------------------------------------------------------------------
```

Где name.aborts - новая процедура обработки ошибок, добавленная в подставку:

```
-- abort, reporting offending item
: name.aborts ( c-addr1 u1 char )
        >R S"  `" $+ R> CHAR-APPEND &' CHAR-APPEND aborts ;
```

Сейчас TINY примет только одну "программу" - пустую(chap10a.frt): :

PROGRAM . (или 'p.' в нашей стенографии).

```
  PROGRAM .   (or 'p.' in our shorthand.)
```

Заметьте, тем не менее, что компилятор генерирует для этой программы корректный код. Она будет выполняться и делать то, что можно ожидать от пустой программы, т.е. ничего

кроме элегантного возвращения в ОС.

Один из моих любимых бенчмарков для компиляторов заключается в компиляции, связывании и выполнении пустой программы для любого языка. Вы можете многое узнать о реализации измеряя предел времени, необходимый для компиляции тривиальной программы. Также интересно измерить количество полученного кода. Во многих компиляторах код может быть довольно большим, потому что они всегда включают целую run-time библиотеку независимо от того, нуждаются они в ней или нет. Ранние версии Turbo Pascal в этом случае производили объектный файл 12K. VAX C генерирует 50K!

Самые маленькие пустые программы какие я видел, получены компиляторами Модула-2 и они занимают примерно 200-800 байт.

В случае TINY у нас еще нет run-time библиотеки, так что объектный код действительно крошечный (tiny): два байта. Это стало рекордом, и вероятно останется таковым, так как это минимальный размер, требуемый ОС.

Следующим шагом будет обработка кода для основной программы. Я буду использовать блок BEGIN из Pascal:

```
<main> ::= BEGIN <block> END
```

Здесь мы снова приняли решение. Мы могли бы потребовать использовать объявление вида "PROCEDURE MAIN", подобно C. Я должен допустить, что это совсем неплохая идея... Мне не особенно нравится подход Паскаля так как я предпочитаю не иметь проблем с определением местоположения основной программы в листинге Паскаля. Но альтернатива тоже немного неудобна, так как вы должны работать с проверкой ошибок когда пользователь опустит основную программу или сделает орфографическую ошибку в ее названии. Здесь я использую простой выход.

Другое решение проблемы "где расположена основная программа" может заключаться в требовании имени для программы и заключения основной программы в скобки:

```
BEGIN <name>
END <name>
```

аналогично соглашению Модула-2. Это добавляет в язык немного "синтаксического сахара". Подобные вещи легко добавлять и изменять по вашим симпатиям если вы сами проектируете язык.

Для синтаксического анализа такого определения основного блока измените процедуру Prog следующим образом:

```
-- --------------------------------------------------------------
-- Parse and translate a program
: prog ( -- )
        'p' match
        header
        main
        '.' match ;
-- --------------------------------------------------------------
```

и добавьте новую процедуру:

```
-- --------------------------------------------------------------
-- Parse and translate a main program
: main ( -- )
        'b' match
        prolog
        'e' match
        epilog ;
-- --------------------------------------------------------------
```

Теперь единственной допустимой программой является программа(chap10b.frt:

```
PROGRAM BEGIN END . (or 'pbe.')
```

Разве мы не делаем успехи??? Хорошо, как обычно это становится лучше. Вы могли бы попробовать сделать здесь некоторые преднамеренные ошибки подобные пропуску 'b' или 'e' и посмотреть что случится. Как всегда компилятор должен отметить все недопустимые входные символы.

ОБЪЯВЛЕНИЯ

Очевидно на следующем шаге необходимо решить, что мы подразумеваем под объявлением. Я намереваюсь иметь два вида объявлений: переменных и процедур/функций. На верхнем уровне разрешены только глобальные объявления, точно как в C.

Сейчас здесь могут быть только объявления переменных, идентифицируемые по ключевому слову VAR (сокращенно "v").

```
<top-level decls> ::= ( <data declaration> )*

<data declaration> ::= VAR <var-list>
```

Обратите внимание, что так как имеется только один тип переменных, нет необходимости объявлять этот тип. Позднее, для полной версии KISS, мы сможем легко добавить описание типа.

Процедура Prog становится:

```
-- ---------------------------------------------------------------
-- Parse and translate a program
: prog ( -- )
        'p' match
        header
        topdecls
        main
        '.' match ;
-- ---------------------------------------------------------------
```

Теперь добавьте две новые процедуры:

```
-- ---------------------------------------------------------------
-- Process a data declaration
: decl ( -- )
        'v' match
        getchar ;
-- ---------------------------------------------------------------
-- Parse and translate global declarations
: topdecls ( -- )
        BEGIN   Look 'b' <>
        WHILE   CASE Look
                 'v' OF decl  ENDOF
                     S" Unrecognized keyword " Look name.aborts
                ENDCASE
        REPEAT ;
-- ---------------------------------------------------------------
```

Заметьте, что на данный момент Decl - просто заглушка. Она не генерирует никакого кода и не обрабатывает список... каждая переменная должна быть в отдельном утверждении VAR.

OK, теперь у нас может быть любое число объявлений данных, каждое начинается с "v" вместо VAR, перед блоком BEGIN. Попробуйте несколько вариантов и посмотрите, что происходит.(chap10c.frt).

ОБЪЯВЛЕНИЯ И ИДЕНТИФИКАТОРЫ

Это выглядит довольно хорошо, но мы все еще генерируем только пустую программу. Настоящий ассемблер должен выдавать директивы ассемблера для распределения памяти под переменные. Пришло время действительно получить какой-нибудь код.

С небольшим дополнительным кодом это легко сделать в процедуре Decl. Измените ее следующим образом:

```
-- ------------------------------------------------------------
-- Parse and translate a data declaration
: decl ( -- )
        'v' match
        getname alloc ;
-- ------------------------------------------------------------
```

Процедура Alloc просто выдает команду ассемблеру для распределения памяти:

```
-- ------------------------------------------------------------
-- Allocate storage for a variable
: alloc ( char -- )
        S" CREATE " ROT CHAR-APPEND S"  0 , " $+ emitdln ;
-- ------------------------------------------------------------
```

Погоняйте программу(chap10d.frt). Попробуйте входную последовательность, которая объявляет какие-нибудь переменные, например:

```
pvxvyvzbe.
```

Видите, как распределяется память? Просто, да? Заметьте также, что точка входа "MAIN" появляется в правильном месте.

Кстати, "настоящий" компилятор имел бы также таблицу идентификаторов для записи используемых переменных. Обычно, таблица идентификаторов необходима для записи типа каждой переменной. Но так как в нашем случае все переменные имеют один и тот же тип, нам не нужна таблица идентификаторов. Оказывается, мы смогли бы находить идентификатор даже без различия типов, но давайте отложим это пока не возникнет такая необходимость.

Конечно, в действительности мы не анализировали правильный синтаксис для объявления данных, так как он включает список переменных. Наша версия разрешает только одну переменную. Это также легко исправить.

БНФ для <var-list> следующая:

```
<var-list> ::= <ident> (, <ident>)*
```

Добавление этого синтаксиса в Decl дает новую версию:

```
-- ------------------------------------------------------------
-- Parse and translate a data declaration
: decl ( -- )
        'v' match
        getname alloc
        BEGIN  Look ',' =
        WHILE  getchar getname alloc
        REPEAT ;
-- ------------------------------------------------------------
```

OK, теперь откомпилируйте этот код и испытайте его(chap10e.frt).. Попробуйте ряд строк с объявлениями VAR, попробуйте список из нескольких переменных в одной строке и комбинации этих двух. Работает?

ИНИЦИАЛИЗАТОРЫ

Пока мы работали с объявлениями данных, меня беспокоила одна вещь - то, что Pascal не позволяет инициализировать данные в объявлении. Эта возможность по общему признанию является своего рода излишеством, и ее может не быть в языке, который считается минимальным языком. Но ее также настолько просто добавить, что было бы позором не сделать этого. БНФ становится:

```
<var-list> ::= <var> ( <var> )*

<var> ::= <ident> [ = <integer> ]
```

Измените Alloc как показано ниже:

```
-- --------------------------------------------------------------
-- Allocate storage for a variable
: alloc ( char -- )
        S" CREATE " ROT CHAR-APPEND BL CHAR-APPEND
        Look '=' = IF   '=' match
                        getnum (.)
                 ELSE   S" 0"
                ENDIF   $+
        S"  ," $+ emitln ;
-- --------------------------------------------------------------
```

Вот оно: инициализатор в шесть дополнительных строк Pascal.

Испытайте эту версию TINY и проверьте, что вы действительно можете задавать начальное значение перменных.(chap10f.frt

Ей богу, он начинает походить на настоящий компилятор! Конечно, он все еще ничего не делает, но выглядит хорошо, не так ли?

Перед тем как оставить этот раздел я должен подчеркнуть, что мы использовали две версии GetNum. Одна, более ранняя, возвращала символьное значение, одиночную цифру. Другая принимала многозначное целое число и возвращала целочисленное значение. Любая из них будет работать здесь, так как WriteLn поддерживает оба типа. Но нет никакой причины ограничивать себя одноразрядными значениями, так что правильной версией для использования будет та, которая возвращает целое число. Вот она:

```
-- --------------------------------------------------------------
-- Get a Number
: getnum ( -- val )
        Look digit? 0= IF  S" Integer" expected  ENDIF
        0 >R
        BEGIN   Look digit?
        WHILE   R> #10 *  Look '0' - +  >R
                getchar
        REPEAT  R> ;
-- --------------------------------------------------------------
```

Строго говоря, мы должны разрешить выражения в поле данных инициализатора, или, по крайней мере, отрицательные значения. Сейчас давайте просто разрешим отрицательные значения изменив код для Alloc следующим образом:

```
-- --------------------------------------------------------------
-- Allocate storage for a variable
: alloc ( char -- )
        S" CREATE " ROT CHAR-APPEND BL CHAR-APPEND
        Look '=' = IF   '=' match
                        Look '-' = IF '-' match S" -" $+ ENDIF
                        getnum (.)
                 ELSE   S" 0"
                ENDIF   $+
```

```
        S"  ," $+ emitln ;

-- ---------------------------------------------------------------
```

Теперь у вас есть возможность инициализировать переменные отрицательными и/или многозначными значениями.(chap10g.frt)

ТАБЛИЦА ИДЕНТИФИКАТОРОВ

Существует одна проблема с компилятором в его текущем состоянии: он ничего не делает для сохранения переменной когда мы ее объявляем. Так что компилятор совершенно спокойно распределит память для нескольких переменных с тем же самым именем. Вы можете легко убедиться в этом набрав строку типа

```
pvavavabe.
```

Здесь мы объявили переменную A три раза. Как вы можете видеть, компилятор бодро принимает это и генерирует три идентичных метки. Не хорошо.

Позднее, когда мы начнем ссылаться на переменные, компилятор также будет позволять нам ссылаться на переменные, которые не существуют. Ассемблер отловит обе эти ошибки, но это совсем не кажется дружественным поведением - передавать такую ошибку ассемблеру. Компилятор должен отлавливать такие вещи на уровне исходного языка.

Так что даже притом, что нам не нужна таблица идентификаторов для записи типов данных, мы должны установить ее только для того, чтобы проверять эти два условия. Так как пока мы все еще ограничены односимвольными именами переменных таблица идентификаторов может быть тривиальной. Чтобы предусмотреть ее сначала добавьте следующее объявление в начало вашей программы:

```
CREATE ST  26 CHARS ALLOT
```

и вставьте следующую функцию:

```
-- ---------------------------------------------------------------
-- Look for symbol in table
: intable ( char -- tf ) 'A' - ST + C@  0<> ;
-- ---------------------------------------------------------------
```

Нам также необходимо инициализировать таблицу пробелами. Следующие строки в Init сделают эту работу:

```
begin
        ST 26 ERASE
   ...
```

Наконец, вставьте следующие две строки в начало Alloc:

```
        ( n) DUP intable IF  S" Duplicate variable name" ROT name.aborts  ENDIF
        'v' OVER 'A' - ST + C!
```

Это должно все решить. Теперь компилятор будет отлавливать двойные объявления. Позднее мы также сможем использовать InTable при генерации ссылок на переменные.(chap10h.frt)

ВЫПОЛНИМЫЕ УТВЕРЖДЕНИЯ

К этому времени мы можем генерировать пустую программу, которая имеет несколько объявленных переменных и возможно инициализированных. Но пока мы не генерировали ни строки выполнимого кода.

Верите ли вы или нет, но мы почти имеем пригодный для использования компилятор! Отсутствует только выполнимый код, который должен входить в основную программу. Но этот код - это только операции присваивания и операторы управления... все вещи, которые мы сделали раньше. Так что у нас не должно занять слишком много времени предусмотреть также и их.

БНФ определение, данное раньше для основной программы, включало операторный блок, который мы пока что игнорировали:

```
<main> ::= BEGIN <block> END
```

Сейчас мы можем рассматривать блок просто как серию операций присваивания:

```
<block> ::= (Assignment)*
```

Давайте начнем с добавления синтаксического анализатора для блока. Мы начнем с процедуры-заглушки для операции присваивания:

```
-- --------------------------------------------------------------
-- Parse and translate an assignment statement
: assignment ( -- ) getchar ;
-- --------------------------------------------------------------
-- Parse and translate a block of statements
: block ( -- )
        BEGIN  Look 'e' <>
        WHILE  assignment
        REPEAT ;
-- --------------------------------------------------------------
```

Измените процедуру Main чтобы она вызывала Block как показано ниже:

```
-- --------------------------------------------------------------
-- Parse and translate a main program
: main ( -- )
        'b' match
        prolog
        block
        'e' match
        epilog ;
-- --------------------------------------------------------------
```

Эта версия все еще не генерирует никакого кода для "операций присваивания"... все что она делает это съедает символы до тех пор, пока не увидит "e", означающее "END". Но она устанавливает основу для того, что следует дальше.

Следующий шаг, конечно, - это расширение кода для операций присваивания. Это то, что мы делали много раз до этого, поэтому я не буду задерживаться на этом. На этот раз, однако, я хотел бы работать с генерацией кода немного по-другому. До настоящего времени мы всегда просто вставляли Emits, которые генерируют выходной код в соответствии с подпрограммами синтасического анализа. Немного неструктурно, возможно, но это кажется самым простым способом и помогает видеть, какой код должен быть выдан для каждой конструкции.

Однако, я понимаю, что большинство из вас используют компьютер 80x86, так что от кода, сгенерированного для 68000 вам мало пользы. Некоторые из вас спрашивали меня, что если бы машинозависимый код мог бы быть собран в одном месте, то было бы проще перенастроить его на другой ЦПУ. Ответ конечно да.

Чтобы сделать это вставьте следующие подпрограммы "генерации кода":

```
-- ------------------------------------------------------------------------------
-- Load/store primary register with a constant/variable
: loadconstant  ( u -- ) (.) S"  d# -> eax  mov," $+ emitln ;
```

```
: loadvariable  ( char -- )
        DUP intable 0= IF  .undefined  ENDIF
        S"  dword-ptr -> eax  mov," ROT CHAR-PREPEND emitln ;

: storevariable ( char -- )
        DUP intable 0= IF  .undefined  ENDIF
        S" eax -> " ROT CHAR-APPEND S"  dword-ptr  mov," $+ emitln ;

-- Allocate storage for a static variable
: allocatestorage ( char c-addr u -- )
        ROT
        S" CREATE " ROT CHAR-APPEND BL CHAR-APPEND
        2SWAP $+ S"  , " $+ emitdln ;

: callfunction  ( char -- )
    S"  offset NEAR call," ROT CHAR-PREPEND emitln ;    -- call a function

: _negate ( -- ) S" eax  neg,"   emitln ;               -- negate primary
: _not    ( -- ) S" eax  not,"   emitln ;               -- not primary
: _push   ( -- ) S" eax  push,"  emitln ;               -- push primary to stack
: _pop    ( -- ) S" eax  pop,"   emitln ;               -- pop primary from stack
: _decr   ( -- ) S" 1 b# -> eax sub," emitln ;          -- decrement primary
: _incr   ( -- ) S" 1 b# -> eax add," emitln ;          -- increment primary
: _decSP  ( -- ) S" [esp -4 +] -> esp lea," emitln ;    -- drop 1 stack item
: _incSP  ( -- ) S" [esp  4 +] -> esp lea," emitln ;    -- restore stack item
: _popadd ( -- )
      S" [esp] -> eax add, [esp 4 +] -> esp lea," emitln ; -- add TOS to primary
: _popsub ( -- )
    S" [esp] -> eax sub, [esp 4 +] -> esp lea," emitln _negate ; -- subtract TOS
from primary
: _popor  ( -- ) S" [esp] -> eax  or,   [esp 4 +] -> esp lea," emitln ;         --
or TOS to primary
: _popxor ( -- ) S" [esp] -> eax  xor,  [esp 4 +] -> esp lea," emitln ;         --
xor TOS to primary
: _popand ( -- ) S" [esp] -> eax  and,  [esp 4 +] -> esp lea," emitln ;         --
xor TOS to primary
: _popcmp ( -- ) S" [esp] -> eax  cmp,  [esp 4 +] -> esp lea," emitln ;         --
compare TOS to primary
: _sete   ( -- ) S" al  sete,   al -> eax  movsx," emitln ;  -- set primary if equal
: _setne  ( -- ) S" al  setne,  al -> eax  movsx," emitln ;  -- set primary if NOT
equal
: _setg   ( -- ) S" al  setl,   al -> eax  movsx," emitln ;  -- set primary if
greater
: _setl   ( -- ) S" al  setg,   al -> eax  movsx," emitln ;  -- set primary if less
: _setge  ( -- ) S" al  setle,  al -> eax  movsx," emitln ;  -- set primary if
greater or equal
: _setle  ( -- ) S" al  setge,  al -> eax  movsx," emitln ;  -- set primary if less
or equal

-- jump always; jump if eax 0
: _branch    ( c-addr u -- ) S"  offset NEAR  jmp," $+ emitln ;
: _pcmp+b0>  ( c-addr u -- ) _popcmp S"  offset NEAR  jg,"  $+ emitln ;
: _branch0   ( c-addr u -- ) S" eax -> eax or,  " 2SWAP $+ S"  offset NEAR  je,"  $+
emitln ;
: _branch<>0 ( c-addr u -- ) S" eax -> eax or,  " 2SWAP $+ S"  offset NEAR  jnz," $+
emitln ;

-- multiply TOS and primary; divide primary by TOS
: _popmul ( -- ) S" [esp] dword mul, [esp 4 +] -> esp lea," emitln ;
: _popdiv ( -- ) S" ecx pop, ecx -> eax xchg, eax -> edx mov, #31 b# -> edx sar, ebx
idiv," emitln ;
-- --------------------------------------------------------------------------
```

Приятная особенность такого подхода, конечно, в том что мы можем перенастроить компилятор на новый ЦПУ просто переписав эти процедуры "генератора кода". Кроме

того, позднее мы обнаружим что можем улучшить качество кода немного подправляя эти процедуры без необходимости изменения компилятора.

Обратите внимание, что и LoadVar и Store проверяют таблицу идентификаторов чтобы удостовериться, что переменная определена. Обработчик ошибки Undefined просто вызывает Abort:

```
-- ------------------------------------------------------------------
-- Report an undefined identifier
: .undefined ( char -- )
   S" Undefined identifier `" ROT CHAR-APPEND &' CHAR-APPEND aborts ;
-- ------------------------------------------------------------------
```

Итак, теперь мы наконец готовы начать обработку выполнимого кода. Мы сделаем это заменив пустую версию процедуры Assignment.

Мы проходили этот путь много раз прежде, так что все это должно быть вам знакомо. Фактически, если бы не изменения, связанные с генерацией кода, мы могли бы просто скопировать процедуры из седьмой части. Так как мы сделали некоторые изменения я не буду их просто копировать, но мы пройдем немного быстрее, чем обычно.

БНФ для операций присваивания:

<assignment> ::= <ident> = <expression>

```
<expression> ::= <first term> ( <addop> <term> )*

<first term> ::= <first factor> <rest>

<term> ::= <factor> <rest>

<rest> ::= ( <mulop> <factor> )*

<first factor> ::= [ <addop> ] <factor>

<factor> ::= <var> | <number> | ( <expression> )
```

Эта БНФ также немного отличается от той, что мы использовали раньше... еще одна "вариация на тему выражений". Эта специфичная версия имеет то, что я считаю лучшей обработкой унарного минуса. Как вы увидите позднее, это позволит нам очень эффективно обрабатывать отрицательные константы. Здесь стоит упомянуть, что мы часто видели преимущества "подстраивания" БНФ по ходу дела, с цель сделать язык легким для анализа. То, что вы видете здесь, немного другое: мы подстраиваем БНФ для того, чтобы сделать генерацию кода более эффективной! Это происходит впервые в этой серии.

Во всяком случае, следующий код реализует эту БНФ:

```
-- -------------------------------------------------------------
-- Parse and Translate a Math Factor

DEFER expression

: factor ( -- )
   Look '(' =  IF  '(' match expression ')' match EXIT  ENDIF
   Look alpha? IF  getname loadvariable
             ELSE  getnum  loadconstant
            ENDIF ;
-- -------------------------------------------------------------
-- Parse and translate a negative factor
: negfactor ( -- )
   '-' match
   Look digit? IF  getnum NEGATE loadconstant EXIT  ENDIF
   factor _negate ;
-- -------------------------------------------------------------
```

```
-- Parse and translate a leading factor
: firstfactor ( -- )
   CASE Look
     '+' OF  '+' match factor ENDOF
     '-' OF   negfactor       ENDOF
              factor
   ENDCASE ;
-- --------------------------------------------------------------
-- Recognize and translate a multiply
: multiply ( -- ) '*' match factor _popmul ;
-- --------------------------------------------------------------
-- Recognize and translate a divide
: divide ( -- ) '/' match factor _popdiv ;
-- --------------------------------------------------------------
-- Common code used by term and firstterm
: term1 ( -- )
   BEGIN Look mulop?
   WHILE _push
         CASE Look
          '*' OF multiply ENDOF
          '/' OF divide   ENDOF
         ENDCASE
   REPEAT ;
-- --------------------------------------------------------------
-- Parse and translate a math term
: term ( -- ) factor term1 ;
-- --------------------------------------------------------------
-- Parse and translate a leading term
: firstterm ( -- ) firstfactor term1 ;
-- --------------------------------------------------------------
-- Recognize and translate an add
: add ( -- ) '+' match term _popadd ;
-- --------------------------------------------------------------
-- Recognize and translate a subtract
: subtract ( -- ) '-' match term _popsub ;
-- --------------------------------------------------------------
-- Parse and translate an expression
:NONAME ( -- )
   firstterm
   BEGIN Look addop?
   WHILE _push
         CASE Look
          '+' OF Add      ENDOF
          '-' OF Subtract ENDOF
         ENDCASE
   REPEAT ; IS expression
-- --------------------------------------------------------------
-- Parse and translate an assignment statement
: assignment ( -- )
   getname LOCAL name
   '=' match
   expression
   name storevariable ;
-- --------------------------------------------------------------
```

ОК, если вы вставили весь этот код, тогда откомпилируйте и проверьте его. Вы должны увидеть приемлемо выглядящий код, представляющий собой законченную программу, которая будет ассемблироваться и выполняться. У нас есть компилятор!(chap10i.frt)

БУЛЕВА ЛОГИКА

Следующий шаг также должен быть вам знаком. Мы должны добавить булевы выражения и операторы отношений. Снова, так как мы работали с ними не один раз, я не буду подробно разбирать их за исключением моментов, в которых они отличаются от того, что мы делали прежде. Снова, мы не будем просто копировать их из других файлов потому что я немного

изменил некоторые вещи. Большинство изменений просто включают изоляцию машинозависимых частей как мы делали для арифметических операций. Я также несколько изменил процедуру NotFactor для соответствия структуре FirstFactor. Наконец я исправил ошибку в объектном коде для операторов отношений: в инструкции Scc я использовал только младшие 8 бит D0. Нам нужно установить логическую истину для всех 16 битов поэтому я добавил инструкцию для изменения младшего байта.

Для начала нам понадобятся несколько подпрограмм распознавания:

```
-- ---------------------------------------------------------------
-- Recognize a boolean Orop
: orop? ( char -- tf ) DUP '|' = SWAP '~' = OR ;
-- ---------------------------------------------------------------
-- Recognize a relop
: relop? ( char -- tf )
        DUP  '=' =
        OVER '#' = OR
        OVER '<' = OR
        SWAP '>' = OR ;
-- ---------------------------------------------------------------
```

Также нам понадобятся несколько подпрограмм генерации кода:

```
{--------------------------------------------------------------}
{ Complement the Primary Register }

procedure NotIt;
begin
   EmitLn('NOT D0');
end;
{--------------------------------------------------------------}
.
.
.
{--------------------------------------------------------------}
{ AND Top of Stack with Primary }

procedure PopAnd;
begin
   EmitLn('AND (SP)+,D0');
end;

{--------------------------------------------------------------}
{ OR Top of Stack with Primary }

procedure PopOr;
begin
   EmitLn('OR (SP)+,D0');
end;

{--------------------------------------------------------------}
{ XOR Top of Stack with Primary }

procedure PopXor;
begin
```

```
   EmitLn('EOR (SP)+,D0');
end;

{--------------------------------------------------------------}
{ Compare Top of Stack with Primary }

procedure PopCompare;
begin
   EmitLn('CMP (SP)+,D0');
end;

{--------------------------------------------------------------}
{ Set D0 If Compare was = }

procedure SetEqual;
begin
   EmitLn('SEQ D0');
   EmitLn('EXT D0');
end;

{--------------------------------------------------------------}
{ Set D0 If Compare was != }

procedure SetNEqual;
begin
   EmitLn('SNE D0');
   EmitLn('EXT D0');
end;

{--------------------------------------------------------------}
{ Set D0 If Compare was > }

procedure SetGreater;
begin
   EmitLn('SLT D0');
   EmitLn('EXT D0');
end;

{--------------------------------------------------------------}
{ Set D0 If Compare was < }

procedure SetLess;
begin
   EmitLn('SGT D0');
   EmitLn('EXT D0');
end;
{--------------------------------------------------------------}
```

Все это дает нам необходимые инструменты. БНФ для булевых выражений такая:

```
<bool-expr> ::= <bool-term> ( <orop> <bool-term> )*

<bool-term> ::= <not-factor> ( <andop> <not-factor> )*

<not-factor> ::= [ '!' ] <relation>
```

```
<relation> ::= <expression> [ <relop> <expression> ]
```

Зоркие читатели могли бы заметить, что этот синтаксис не включает нетерминал "bool-factor" используемый в ранних версиях. Тогда он был необходим потому, что я также разрешал булевы константы TRUE и FALSE. Но не забудьте, что в TINY нет никакого различия между булевыми и арифметическими типами... они могут свободно смешиваться. Так что нет нужды в этих предопределенных значениях... мы можем просто использовать -1 и 0 соответственно.

В терминологии C мы могли бы всегда использовать определения:

```
#define TRUE -1
#define FALSE 0
```

(Так было бы, если бы TINY имел препроцессор.) Позднее, когда мы разрешим объявление констант, эти два значения будут предопределены языком.

Причина того, что я заостряю на этом ваше внимание, в том что я пытался использовать альтернативный путь, который заключался в использовании TRUE и FALSE как ключевых слов. Проблема с этим подходом в том, что он требует лексического анализа каждого имени переменной в каждом выражении. Как вы помните, я указал в главе 7, что это значительно замедляет компилятор. Пока ключевые слова не могут быть в выражениях нам нужно выполнять сканирование только в начале каждого нового оператора... значительное улучшение. Так что использование вышеуказанного синтаксиса не только упрощает синтаксический анализ, но также ускоряет сканирование.

Итак, если мы удовлетворены синтаксисом, представленным выше, то соответствующий код показан ниже:

```
-- ----------------------------------------------------------------
-- Recognize and translate a relational "equals"
: equals ( -- ) '=' match expression _popcmp _sete ;
-- ----------------------------------------------------------------
-- Recognize and translate a relational "not equals"
: notequals ( -- ) '#' match expression _popcmp _setne ;
-- ----------------------------------------------------------------
-- Recognize and translate a relational "less than"
: less ( -- ) '<' match expression _popcmp _setl ;
-- ----------------------------------------------------------------
-- Recognize and translate a relational "greater than"
: greater ( -- ) '>' match expression _popcmp _setg ;
-- ----------------------------------------------------------------
-- Parse and translate a relation
: relation ( -- )
   expression
   Look relop? 0= ?EXIT
   _push
   CASE Look
    '=' OF equals    ENDOF
    '#' OF notequals ENDOF
    '<' OF less      ENDOF
    '>' OF greater   ENDOF
   ENDCASE ;
-- ----------------------------------------------------------------
-- Parse and translate a boolean factor with leading NOT
: notfactor ( -- )
   Look '!' <> IF  relation  EXIT  ENDIF
   '!' match  relation _not ;
-- ----------------------------------------------------------------
-- Parse and translate a boolean term
: boolterm ( -- )
   notfactor
   BEGIN  Look '&' =
```

```
   WHILE  _push  '&' match  notfactor  _popand
   REPEAT ;
-- ---------------------------------------------------------------
-- Recognize and translate a boolean OR
: boolOR ( -- ) '|' match boolterm _popor ;
-- ---------------------------------------------------------------
-- Recognize and translate an EXCLUSIVE OR
: boolXOR ( -- ) '~' match boolterm _popxor ;
-- ---------------------------------------------------------------
-- Parse and translate a boolean expression
:NONAME ( -- )
   boolterm
   BEGIN Look orop?
   WHILE _push
         CASE Look
          '|' OF boolor ENDOF
          '~' OF boolxor ENDOF
         ENDCASE
   REPEAT ; IS boolexpression
-- ---------------------------------------------------------------
```

Чтобы связать все это вместе не забудьте изменить обращение к Expression в процедурах Factor и Assignment на вызов BoolExpression.

Хорошо, если вы набрали все это, откомпилируйте и погоняйте эту версию.(chap10j.frt) Сначала удостоверьтесь, что вы все еще можете анализировать обычные арифметические выражения. Затем попробуйте булевские. Наконец удостоверьтесь, что вы можете присваивать результат сравнения. Попробуйте к примеру:

```
pvx,y,zbx=z>ye.
```

что означает

```
PROGRAM
VAR X,Y,Z
BEGIN
X = Z > Y
END.
```

Видите как происходит присваивание булевского значения X?

УПРАВЛЯЮЩИЕ СТРУКТУРЫ

Мы почти дома. Имея булевы выражения легко добавить управляющие структуры. Для TINY мы разрешим только две из них, IF и WHILE:

```
<if> ::= IF <bool-expression> <block> [ ELSE <block>] ENDIF

<while> ::= WHILE <bool-expression> <block> ENDWHILE
```

Еще раз позвольте мне разъяснить решения, подразумевающиеся в этом синтаксисе, который сильно отличается от синтаксиса C или Pascal. В обоих этих языках "тело" IF или WHILE расценивается как одиночный оператор. Если вы предполагаете использовать блок из более чем одного оператора вы должны создать составной утверждение использую BEGIN-END (в Pascal) или '{}' (в C). В TINY (и KISS) нет таких вещей как составное утверждение... одиночное или множественное, они являются в этом языке просто блоками.

В KISS все управляющие структуры имеют явные и уникальные ключевые слова, выделяющие операторный блок поэтому не может быть никакой путаницы где он начинается и заканчивается. Это современный подход, используемый в таких уважаемых языках, как Ada и Modula-2 и он полностью устраняет проблему "висячих else".

Обратите внимание, что я мог бы использовать то же самое ключевое слово END для

завершения всех конструкций, как это сделано в Pascal. (Закрывающая '}' в C служит той же самой цели.) Но это всегда вело к неразберихе, вот почему программисты на Pascal предпочитают писать так:

```
end { loop }
```

или

```
end { if }
```

Как я объяснил в пятой части, использование уникальных терминальных ключевых слов увеличивает размер списка ключевых слов и, следовательно, замедляет лексический анализ, но в данном случае это кажется небольшой ценой за дополнительную подстраховку. Лучше обнаруживать ошибки во время компиляции, чем во время выполнения.

Одна последняя мысль: каждая из двух конструкций выше имеют нетерминалы

```
<bool-expression> and <block>
```

расположенные рядом без разделяющих ключевых слов. В Паскале мы ожидали бы в этом месте ключевые слова THEN и DO.

Я не вижу проблем в том, чтобы опустить эти ключевые слова, и синтаксический анализатор также не будет иметь проблем, при условии, что мы не сделаем ошибок в bool-expression. С другой стороны, если мы включим эти дополнительные ключевые слова мы получили бы еще один уровень подстраховки за малые деньги, и с этим у меня также нет проблем. Примите правильное решение каким путем пойти.

OK, после этого небольшого объяснения давайте продолжим. Как обычно нам понадобятся несколько новых подпрограмм генерации кода. Они генерируют код для условных и безусловных переходов:

```
-- ----------------------------------------------------------------
-- Recognize and translate an IF Construct

DEFER block ( -- )

: doIF ( -- )
        0 0 LOCALS| L2 L1 |
        'i' match
        boolexpression
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L1 PACK DROP
        L1 COUNT DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP
        L1 COUNT _branch0
        block
        Look 'l'  = IF  L2 FREE ?ALLOCATE
                        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP
                        L2 COUNT _branch
                        L1 COUNT postlabel
                        block
                 ENDIF
        L2 COUNT postlabel
        'e' match
        L1 FREE ?ALLOCATE
        L2 FREE ?ALLOCATE ;
-- ---------------------------------------------------------------
-- Parse and translate a WHILE Statement
: doWHILE ( -- )
        0 0 LOCALS| L2 L1 |
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L1 PACK DROP
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP
        'w' match
        L1 COUNT postlabel
```

```
        boolexpression
        L2 COUNT _branch0
        block
        'e' match
        L1 COUNT _branch
        L2 COUNT postlabel
        L1 FREE ?ALLOCATE
        L2 FREE ?ALLOCATE ;
-- ---------------------------------------------------------------
```

Исключая изоляцию подпрограмм генератора кода, код для анализа управляющих конструкций такой же, как вы видели прежде:

```
{--------------------------------------------------------------}
{ Recognize and Translate an IF Construct }

procedure Block; Forward;

procedure DoIf;
var L1, L2: string;
begin
   Match('i');
   BoolExpression;
   L1 := NewLabel;
   L2 := L1;
   BranchFalse(L1);
   Block;
   if Look = 'l' then begin
      Match('l');
      L2 := NewLabel;
      Branch(L2);
      PostLabel(L1);
      Block;
   end;
   PostLabel(L2);
   Match('e');
end;

{--------------------------------------------------------------}
{ Parse and Translate a WHILE Statement }

procedure DoWhile;
var L1, L2: string;
begin
   Match('w');
   L1 := NewLabel;
   L2 := NewLabel;
   PostLabel(L1);
   BoolExpression;
   BranchFalse(L2);
   Block;
   Match('e');
```

```
  Branch(L1);
  PostLabel(L2);
end;
{--------------------------------------------------------------}
```

Чтобы связать все это вместе нам нужно только изменить процедуру Block чтобы распознавать ключевые слова IF и WHILE. Как обычно мы расширим определение блока так:

```
<block> ::= ( <statement> )*
```

где

```
<statement> ::= <if> | <while> | <assignment>
```

Соответствующий код:

```
-- ----------------------------------------------------------------
-- Parse and translate a block of statements
:NONAME BEGIN  Look 'e' <>
               Look 'l' <> AND
        WHILE  CASE Look
                'i' OF  doIF      ENDOF
                'w' OF  doWHILE   ENDOF
                        assignment
               ENDCASE
        REPEAT ; IS block
-- ----------------------------------------------------------------
```

Добавьте подпрограммы, которые я дал, откомпилируйте и протестируйте их(chap10k.frt). У вас должна быть возможность анализировать односимвольные версии любых управляющих конструкции. Выглядит довольно хорошо!

Фактически, за исключением односимвольного ограничения, мы получили практически полную версию TINY. Я назову его TINY Version 0.1.

ЛЕКСИЧЕСКИЙ АНАЛИЗ

Конечно, вы знаете, что будет дальше: Мы должны преобразовать программу так, чтобы она могла работать с многосимвольными ключевыми словами, переводами строк и пробелами. Мы только что прошли все это в седьмой главе. Мы будем использовать метод распределенного сканера, который я показал вам в этой главе. Фактическая реализация немного отличается, потому что различается способ, которым я обрабатываю переводы строк.

Для начала, давайте просто разрешим пробелы. Для этого необходимо только добавить вызовы SkipWhite в конец трех подпрограмм GetName, GetNum и Match. Вызов SkipWhite в Init запускает помпу в случае если есть ведущие пробелы.

Затем мы должны обрабатывать переводы строк. Это в действительности двухшаговый процесс так как обработка переносов с односимвольными токенами отличается от таковой для многосимвольных токенов. Мы можем устранить часть работы сделав оба шага одновременно, но я чувствую себя спокойней, работая последовательно.

Вставьте новую процедуру:

```
-- ----------------------------------------------------------------
-- Skip over an end-of-line
: newline ( -- )
   BEGIN Look ^M =
   WHILE getchar
         Look ^J = IF  getchar  ENDIF
         skipwhite
   REPEAT ;
```

```
-- --------------------------------------------------------------
```

Заметьте, что мы видели эту процедуру раньше в виде процедуры Fin. Я изменил имя, так как новое кажется более соответствующим фактическому назначению. Я также изменил код чтобы учесть множественные переносы и строки только с пробелами.

Следующим шагом будет вставка вызовов NewLine везде, где мы посчитаем перенос допустимым. Как я подчеркивал ранее, этот момент может очень различаться для разных языков. В TINY я решил разрешить их практически в любом месте. Это означает, что нам нужно вызывать NewLine в начале (не в конце как с SkipWhite) процедур GetName, GetNum и Match.

Для процедур, которые имеют циклы While, таких как TopDecl, нам нужен вызов NewLine в начале процедуры и в конце каждого цикла. Таким способом мы можем быть уверены, что NewLine вызывается в начале каждого прохода через цикл.

Если вы все это сделали, испытайте программу и проверьте, что она действительно обрабатывает пробелы и переносы.

Если это так, тогда мы готовы работать с многосимвольными токенами и ключевыми словами. Для начала, добавьте дополнительные объявления (скопированные почти дословно из главы 7):

```
-- --------------------------------------------------------------
-- Type declarations
  8 =: /symbol
100 =: maxentry
-- --------------------------------------------------------------
-- Variable declarations
0 VALUE token                  -- encoded token
CREATE token$ 16 CHARS ALLOT   -- unencoded token
-- --------------------------------------------------------------
-- Definition of keywords and token types
-- helper to build symbol tables
: $, ( c-addr u size -- )
        >S S MIN >R
        R@ C, R@ 0 ?DO  C@+ C,  LOOP DROP
        S> R> - CHARS ALLOT ;
-- Symboltable builder itself
: SYMTAB ( size -- )
        CREATE  DUP 0< IF  0 , ABS DUP ,
                           1+ maxentry * ALLOT
                     ELSE  HERE >R
                           0 ,  ( #items )
                           DUP >S ,  ( itemsize )
                           BEGIN  BL <WORD> DUP
                           WHILE  S $, 1 R@ +!
                           REPEAT 2DROP -R -S
                    ENDIF
        DOES>   CELL+ @+ 1+ ROT * + ( ix -- addr ) ;

/symbol SYMTAB KWlist IF ELSE ENDIF WHILE ENDWHILE VAR BEGIN END PROGRAM

: KW->token ( kw_index -- ) 2+ C" xilewevbep" + C@ TO token ;
-- --------------------------------------------------------------
```

Затем добавьте три процедуры, также из седьмой главы:

```
-- --------------------------------------------------------------
-- Table Lookup
: lookup ( c-addr u 'table -- n2 | -1 )
        0 0 LOCALS| /symbol n table sz addr |
        table 2 CELLS - @+ TO n  @ TO /symbol
        n 0<= IF  -1 EXIT  ENDIF
           0 n DO
```

```
                  /symbol 1+ I * table + COUNT
                  addr sz COMPARE 0= IF  I UNLOOP EXIT  ENDIF
         -1 +LOOP
        -1 ;
-- ---------------------------------------------------------------
.
.
-- ---------------------------------------------------------------
-- Get an identifier and scan it for keywords
: scan ( -- ) getname 0 KWlist lookup KW->token ;
-- ---------------------------------------------------------------
-- Match a specific input string
: matchstring ( c-addr u -- )
        token$ COUNT 2OVER
        COMPARE IF  &` CHAR-PREPEND &' CHAR-APPEND expected
              ELSE  2DROP
             ENDIF ;
-- ---------------------------------------------------------------
```

Теперь мы должны сделать довольно много тонких изменений в оставшихся процедурах. Сначала мы должны изменить функцию GetName на процедуру, снова как в главе 7:

```
-- ---------------------------------------------------------------
-- Get an identifier
: getname ( -- c-addr u )
        newline
        Look alpha? 0= IF  S" Name" expected  ENDIF
        token$ C0!
        BEGIN   Look alnum?
        WHILE   Look >UPC token$ char+!  getchar
        REPEAT
        token$ COUNT  skipwhite ;
-- ---------------------------------------------------------------
```

Обратите внимание, что эта процедура оставляет свой результат в глобальной строковой переменной Value.

Затем, мы должны изменить каждую обращение к GetName чтобы отразить ее новую форму. Они происходят в Factor, Assignment и Decl:

```
-- ----------------------------------------------------------------
-- Parse and translate a math factor

DEFER boolexpression ( -- )

: factor ( -- )
        Look '('  = IF  '(' match  boolexpression  ')' match EXIT  ENDIF
        Look alpha? IF  getname DROP C@ loadvariable         EXIT  ENDIF
        getnum loadconstant ;
-- ---------------------------------------------------------------
.
.
-- ---------------------------------------------------------------
-- Parse and translate an assignment statement
: assignment ( -- )
        token$ COUNT DUP 1+ ALLOCATE ?ALLOCATE DUP LOCAL name PACK DROP
        '=' match
        boolexpression
        name COUNT DROP C@ storevariable
        name FREE ?ALLOCATE ;
-- ---------------------------------------------------------------
.
.
-- ---------------------------------------------------------------
-- Parse and translate a data declaration
: decl ( -- )
```

```
        getname DROP C@ alloc
        BEGIN  Look ',' =
        WHILE  ',' match  getname DROP C@ alloc
        REPEAT ;
-- ---------------------------------------------------------------
```

(Заметьте, что мы все еще разрешаем только односимвольные имена переменных поэтому мы используем здесь простое решение и просто используем первый символ строки.)

Наконец, мы должны внести изменения, позволяющие использовать Token вместо Look как символа для проверки и вызывать Scan в подходящих местах. По большей части это включает удаление вызовов Match, редкие замены вызовов Match на вызовы MatchString, и замену вызовов NewLine на вызовы Scan. Вот затронутые подпрограммы:

```
-- ----------------------------------------------------------------
-- Recognize and translate an IF construct

DEFER block

: doIF ( -- )
        0 0 LOCALS| L2 L1 |
        boolexpression
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L1 PACK DROP
        L1 COUNT DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP
        L1 COUNT _branch0
        block
        token 'l' = IF  L2 FREE ?ALLOCATE
                        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP
                        L2 COUNT _branch
                        L1 COUNT postlabel
                        block
                 ENDIF
        L2 COUNT postlabel
        S" ENDIF" matchstring

        L1 FREE ?ALLOCATE
        L2 FREE ?ALLOCATE ;
-- ---------------------------------------------------------------
-- Parse and translate a WHILE statement
: doWHILE ( -- )
        0 0 LOCALS| L2 L1 |
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L1 PACK DROP
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP

        L1 COUNT postlabel
        boolexpression
        L2 COUNT _branch0
        block
        L1 COUNT _branch
        L2 COUNT postlabel
        S" ENDWHILE" matchstring

        L1 FREE ?ALLOCATE
        L2 FREE ?ALLOCATE ;
-- ---------------------------------------------------------------
-- Parse and translate a block of statements
:NONAME ( label -- )
        LOCAL L
        scan
        BEGIN  token 'e' <>
               token 'l' <> AND
        WHILE  CASE token
                'i' OF  doIF      ENDOF
                'w' OF  doWHILE   ENDOF
                        assignment
               ENDCASE
```

```
            scan
        REPEAT ; IS block
-- -------------------------------------------------------------
-- Parse and Translate Global Declarations
: topdecls ( -- )
        scan
        BEGIN   token 'b' <>
        WHILE   CASE token
                 'v' OF decl  ENDOF
                     S" Unrecognized keyword "
                     token$ COUNT name.aborts
                ENDCASE
                scan
        REPEAT ;
-- -------------------------------------------------------------
-- Parse and translate a main program
: doprogram ( -- )
        S" BEGIN" matchstring
        prolog
        block
        S" END" matchstring
        epilog ;
-- -------------------------------------------------------------
-- Parse and translate a program
: prog ( -- )
        S" PROGRAM" matchstring
        getname prog$ PACK DROP
        header
        topdecls
        doprogram
        '.' match ;
-- -------------------------------------------------------------
-- Initialize
: init ( -- )
        CLEAR Lcount
        ST 26 CHARS ERASE
        CR getchar scan ;
-- -------------------------------------------------------------
```

Это должно работать. Если все изменения сделаны правильно, вы должны теперь анализировать программы, которые выглядят как программы. (Если вы не сделали всех изменений, не отчаивайтесь. Полный листинг конечной формы дан ниже.)

Работает? Если да, то мы почти дома. Фактически, с несколькими небольшими исключениями, мы уже получили компилятор, пригодный для использования. Имеются еще несколько областей, требующих усовершенствования.

МНОГОСИМВОЛЬНЫЕ ИМЕНА ПЕРЕМЕННЫХ

Одна из них - ограничение, требующее использования односимвольных имен переменных. Теперь, когда мы можем обрабатывать многосимвольные ключевые слова, это ограничение начинает казаться произвольным и ненужным. И действительно это так. В основном, единственное его достоинство в том, что он позволяет получить тривиально простую реализацию таблицы идентификаторов. Но это просто удобство для создателей компиляторов и оно должно быть уничтожено.

Мы уже делали этот шаг прежде. На этот раз, как обычно, я сделаю это немного по-другому. Я думаю подход, примененный здесь, сохранит простоту настолько, насколько это возможно.

Естественным путем реализации таблицы идентификаторов на Pascal является объявление переменной типа запись и создание таблицы идентификаторов как массива таких записей. Здесь, однако, нам в действительности пока не нужно поле типа (существует пока что только один разрешенный тип), так что нам нужен только массив символов. Это имеет

свое преимущество, потому что мы можем использовать существующую процедуру Lookup для поиска в таблице идентификаторов также как и в списке ключевых слов. Оказывается, даже когда нам нужны больше полей, мы все равно можем использовать тот же самый подход, просто сохраняя другие поля в отдельных массивах.

Вот изменения, которые необходимо сделать. Сперва добавьте новую типизированную константу:

```
      0 ST    2 CELLS - =: [cnt]ST
      0 SType 2 CELLS - =: [cnt]SType
```

Затем измените определение таблицы идентификаторов как показано ниже:

```
100 =: maxentry
.
.
/symbol  NEGATE SYMTAB ST
=CELL 1- NEGATE SYMTAB SType
```

(Обратите внимание, что ST не объявлен как SymTab. Это объявление липовое, чтобы заставить Lookup работать. SymTab заняля бы слишком много памяти и поэтому фактически никогда не обьявляется).

Затем мы должны заменить InTable.

```
-- ---------------------------------------------------------------
-- Look for symbol in table
: intable ( c-addr u -- tf ) 0 ST lookup 0>= ;
-- ---------------------------------------------------------------
```

Нам также необходима новая процедура AddEntry, которая добавляет новый элемент в таблицу:

```
-- ---------------------------------------------------------------
-- Add a new entry to the symbol table
: addentry ( c-addr u type -- )
        [cnt]ST LOCALS| #entries T sz addr |
        addr sz intable IF  S" Duplicate identifier " addr sz name.aborts  ENDIF
        #entries @ maxentry = IF  S" Symbol table full" aborts  ENDIF
        1 #entries +!
        addr sz  #entries @ ST    PACK DROP
        T        #entries @ SType C! ;
-- ---------------------------------------------------------------
```

Эта процедура вызывается из Alloc:

```
-- ---------------------------------------------------------------
-- Allocate storage for a variable
: alloc ( c-addr u -- )
        LOCALS| sz addr |
        addr sz intable IF  S" Duplicate variable name "
                            addr sz name.aborts
                     ENDIF
        addr sz 'v' addentry
.
.
.
-- ---------------------------------------------------------------
```

Наконец, мы должны изменить все подпрограммы, которые в настоящее время обрабатывают имена переменных как одиночный символ. Они включают LoadVar и Store (просто измените тип с char на string) и Factor, Assignment и Decl (просто измените Value[1]

на Value).

Последняя вещь: измените процедуру Init для очистки массива как показано ниже:

```
-- ---------------------------------------------------------------
-- Initialize
: init ( -- )
        CLEAR Lcount
        [cnt]ST    OFF
        [cnt]SType OFF
        CR getchar scan ;
-- ---------------------------------------------------------------
```

Это должно работать. Испытайте ее и проверьте, что вы действительно можете использовать многосимвольные имена переменных.

СНОВА ОПЕРАТОРЫ ОТНОШЕНИЙ

У нас осталось последнее односимвольное ограничение - ограничение операторов отношений. Некоторые из операторов отношений действительно состоят из одиночных символов, но другие требуют двух. Это '<=' и '>='. Я также предпочитаю Паскалевское '<>' для "не равно" вместо '#'.

Как вы помните, в главе 7 я указал, что стандартный способ работы с операторами отношений - включить их в список ключевых слов и позволить лексическому анализатору отыскивать их. Но, опять, это требует выполнение полного анализа выражения, тогда как до этого мы у нас была возможность ограничить использование сканера началом утверждения.

Я упомянул тогда, что мы все же можем избежать неприятностей с этим, так как многосимвольных операторов отношений немного и они ограничены в применении. Было бы легко обрабатывать их просто как специальные случаи и поддерживать их специальным способом.

Требуемые изменения влияют только на подпрограммы генерации кода и процедуры Relation и ее друзей. Сперва, нам понадобятся еще две подпрограммы генерации кода:

```
{---------------------------------------------------------------}
{ Set D0 If Compare was <= }

procedure SetLessOrEqual;
begin
   EmitLn('SGE D0');
   EmitLn('EXT D0');
end;

{---------------------------------------------------------------}
{ Set D0 If Compare was >= }

procedure SetGreaterOrEqual;
begin
   EmitLn('SLE D0');
   EmitLn('EXT D0');
end;
{---------------------------------------------------------------}
```

Затем измените подпрограммы анализа отношений как показано ниже:

```
-- ---------------------------------------------------------------
-- Recognize and translate a relational "less than or equal"
: lessorequal ( -- )  '=' match  expression _popcmp _setle ;
```

```
-- --------------------------------------------------------------
-- Recognize and Translate a Relational "Not Equals"
: notequals   ( -- )  '>' match  expression _popcmp _setne ;
-- --------------------------------------------------------------
-- recognize and translate a relational "less than"
: less ( -- )
        '<' match
        Look '=' = IF  lessorequal EXIT  ENDIF
        Look '>' = IF  notequals   EXIT  ENDIF
        expression _popcmp _setl ;
-- --------------------------------------------------------------
-- recognize and translate a relational "greater than"
: greater ( -- )
        '>' match
        Look '=' = IF   '=' match expression _popcmp _setge
                        EXIT
                ENDIF
        expression _popcmp _setg ;
-- --------------------------------------------------------------
```

Это все, что требуется. Теперь вы можете обрабатывать все операторы отношений. Попробуйте.

ВВОД/ВЫВОД

Теперь у нас есть полный, работающий язык, за исключением одного небольшого смущающего факта: у нас нет никакого способа получить или вывести данные. Нам нужны подпрограммы ввода/вывода.

Современное соглашение, установленное в C и продолженное в Ada и Modula-2, состоит в том, чтобы вывести I/O операторы из самого языка и просто включить их в библиотеку подпрограмм. Это было бы прекрасно, за исключением того, что мы пока не имеем никаких средств поддержки подпрограмм. В любом случае, с этим подходом вы столкнетесь с проблемой переменной длины списка параметров. В Паскале I/O операторы встроены в язык, поэтому это единственные операторы, для которых список параметров может иметь переменное число элементов. В C мы примиряемся с клуджами типа scanf и printf и должны передавать количество параметров в вызываемую процедуру. В Ada и Modula-2 мы должны использовать неудобный (и медленный!) способ отдельного вызова для каждого аргумента.

Так что я думаю, что предпочитаю Паскалевский подход встраивания подпрограмм ввода/вывода, даже если мы не нуждаемся в этом.

Как обычно, для этого нам нужны еще несколько подпрограмм генерации кода. Они, оказывается, самые простые из всех, потому что все, что мы делаем это вызываем библиотечные процедуры для выполнения работы.

```
-- --------------------------------------------------------------
-- read to primary register and store in variable
: readvar  ( c-addr u -- ) S" READ" callfunction  storevariable ;
-- --------------------------------------------------------------
-- write from primary register
: writevar ( -- ) S" WRITE" callfunction ;
```

Идея состоит в том, что READ загружает значение из входного потока в D0, а WRITE выводит его оттуда.

Эти две процедуры представляют собой нашу первую встречу с потребностью в библиотечных процедурах... компонентах Run Time Library (RTL). Конечно кто-то (а именно мы) должен написать эти подпрограммы, но они не являются непосредственно частью компилятора. Я даже не буду беспокоиться о том, чтобы показать здесь эти подпрограммы, так как они очевидно очень ОС-зависимы. Я просто скажу, что для SK*DOS они особенно просты... почти тривиальны. Одна из причин, по которым я не буду показывать их здесь в том,

что вы можете добавлять новые виды возможностей, например приглашение в READ или возможность пользователю повторить ошибочный ввод.

Но это действительно отдельный от компилятора проект, так что теперь я буду подразумевать что библиотека, называемая TINYLIB.LIB, существует.

Так как нам теперь нужно загружать ее, мы должны добавить ее загрузку в процедуру Header:

```
-- --------------------------------------------------------------
-- Write header info
: header ( -- )
        S" -- DATA section --------" emitdln
        S" NEEDS -tinylib" emitdln
        S" " emitdln ;
-- --------------------------------------------------------------
```

Она возьмет на себя эту часть работы. Теперь нам также необходимо распознавать команды ввода и вывода. Мы можем сделать это добавив еще два ключевых слова в наш список:

```
-- --------------------------------------------------------------
-- Definition of keywords and token types
/symbol SYMTAB KWlist IF ELSE ENDIF WHILE ENDWHILE READ WRITE VAR BEGIN END PROGRAM

: KW->token ( kw_index -- ) 2+ C" xileweRWvbep" + C@ TO token ;
-- --------------------------------------------------------------
```

(Обратите внимание, что здесь я использую кода в верхнем регистре чтобы избежать конфликта с 'w' из WHILE.)

Затем нам нужны процедуры для обработки оператора ввода/вывода и его списка параметров:

```
-- --------------------------------------------------------------
-- Process a read statement
: doread ( -- )
        '(' match
        getname readvar
        BEGIN  Look ',' =
        WHILE  ',' match getname readvar
        REPEAT
        ')' match ;
-- ---------------------------------------------------------------
-- Process a write statement
: dowrite ( -- )
        '(' match
        expression writevar
        BEGIN  Look ',' =
        WHILE  ',' match expression writevar
        REPEAT
        ')' match ;
-- --------------------------------------------------------------
```

Наконец, мы должны расширить процедуру Block для поддержки новых типов операторов:

```
-- --------------------------------------------------------------
-- Parse and translate a block of statements
:NONAME ( -- )
        scan
        BEGIN  token 'e' <>
               token 'l' <> AND
        WHILE  CASE token
                'i' OF  doIF      ENDOF
                'w' OF  doWHILE   ENDOF
```

```
             'R' OF  doread    ENDOF
             'W' OF  dowrite   ENDOF
                     assignment
            ENDCASE
            scan
        REPEAT ; IS block
-- ---------------------------------------------------------------
```

На этом все. Теперь у нас есть язык!

ЗАКЛЮЧЕНИЕ

К этому моменту мы полностью определили TINY. Он не слишком значителен... в действительности игрушечный комиплятор. TINY имеет только один тип данных и не имеет подпрограмм... но это законченный, пригодный для использования язык. Пока что вы не имеете возможности написать на нем другой компилятор или сделать что-нибудь еще очень серьезное, но вы могли бы писать программы для чтения входных данных, выполнения вычислений и вывода результатов. Не слишком плохо для игрушки.

Более важно, что мы имеем твердую основу для дальнейшего развития. Я знаю, что вы будете рады слышать это: в последний раз я начал с создания синтаксического анализатора заново... с этого момента я предполагаю просто добавлять возможности в TINY пока он не превратится в KISS. Ох, будет время, когда нам понадобится попробовать некоторые вещи с новыми копиями Cradle, но как только мы разузнаем как они делаются, они будут встроены в TINY.

Какие это будут возможности? Хорошо, для начала нам понадобятся подпрограммы и функции. Затем нам нужна возможность обрабатывать различные типы, включая массивы, строки и другие структуры. Затем нам нужно работать с идеей указателей. Все это будет в следующих главах.

Увидимся.

В справочных целях полный листинг TINY версии 1.0 показан ниже(tiny10.frt): :

```
-- Variable declarations
-------------------------------------------------------------------------------------

-1 =: unknown

0 VALUE Lcount                          -- label counter
0 VALUE Look                            -- lookahead character
0 VALUE token                           -- encoded token

#100 =: maxentry

CREATE token$  0 C, #256 CHARS ALLOT    -- unencoded token
CREATE prog$   0 C, #256 CHARS ALLOT    -- program name

-- Symbol tables and Labels
------------------------------------------------------------------------------------

-- helper to build symbol tables
: $, ( c-addr u size -- )
        >S S MIN >R
        R@ C, R@ 0 ?DO  C@+ C,  LOOP DROP
        S> R> - CHARS ALLOT ;

-- Type declarations
: SYMTAB ( size -- )
        CREATE  DUP 0< IF  0 , ABS DUP ,
                           1+ maxentry * ALLOT
                     ELSE  HERE >R
                           0 ,  ( #items )
                           DUP >S ,  ( itemsize )
```

```
                            BEGIN  BL <WORD> DUP
                            WHILE  2DUP S" \" COMPARE
                                   0= IF  2DROP REFILL DROP
                                    ELSE  S $, 1 R@ +!
                                   ENDIF
                            REPEAT 2DROP -R -S
                     ENDIF
        DOES>   CELL+ @+ 1+ ROT * + ( ix -- addr ) ;

-- Definition of keywords and token types

8 CHARS =: /symbol

/symbol SYMTAB KWlist   IF ELSE ENDIF   \
                        WHILE ENDWHILE  \
                        DO ENDDO        \
                        LOOP ENDLOOP    \
                        REPEAT UNTIL    \
                        FOR TO ENDFOR   \
                        BREAK           \
                        READ WRITE      \
                        VAR BEGIN END PROGRAM

: KW->token ( kw_index -- ) 2+ C" xilewedeLerufteBRWvbep" + C@ TO token ;

        /symbol  NEGATE SYMTAB ST
        =CELL 1- NEGATE SYMTAB SType

0 ST    2 CELLS - =: [cnt]ST
0 SType 2 CELLS - =: [cnt]SType

: lookup ( c-addr u 'table -- n2|unknown )
        0 0 LOCALS| /symbol n table sz addr |
        table 2 CELLS - @+ TO n  @ TO /symbol
        n 0<= IF  unknown EXIT  ENDIF
           0 n DO
                 /symbol 1+ I * table + COUNT
                 addr sz COMPARE 0= IF  I UNLOOP EXIT  ENDIF
         -1 +LOOP
        unknown ;

-- Locate a symbol in table
-- Returns the index of the entry.  (unknown) -1 if not present.
: locate ( c-addr u -- ix ) 0 ST lookup ;

-- Look for symbol in table
: intable ( c-addr u -- tf )
        0 ST lookup 0>= ;

-- Dump the symbol tabel
: .symbols ( -- )
        [cnt]ST @ 0= IF  CR ." No symbols defined." EXIT  ENDIF
        CR ." -- type --.-------- name ---------"
        CR
        [cnt]ST @ 0 ?DO  CR   2 HTAB   I 1+ SType C@ EMIT
                             #14 HTAB I 1+ ST    .$
                  LOOP ;

-- Tools
------------------------------------------------------------------------------------

: getchar  ( -- )          EKEY TO Look ;                      -- read new
character from input stream
: error    ( c-addr u -- ) CR ^G EMIT ." Error: " TYPE ." ." ;  -- report an error
: aborts   ( c-addr u -- ) error ABORT ;                       -- report error and
halt
```

```
: expected ( c-addr u -- ) S"  Expected" $+ aborts ;             -- report what was
expected
: alpha?   ( char -- tf )  >UPC 'A' 'Z' 1+ WITHIN ;              -- recognize an
alpha character
: digit?   ( char -- tf )  '0' '9' 1+ WITHIN ;                   -- recognize a
decimal digit
: alnum?   ( char -- tf )  DUP alpha? SWAP digit? OR ;           -- recognize
alphanumeric
: white?   ( char -- tf )  DUP Tab =  SWAP BL  =  OR ;           -- recognize white
space
: orop?    ( char -- tf )  DUP '|' =  SWAP '~' =  OR ;           -- recognize an OR
operand
: addop?   ( char -- tf )  DUP '+' =  SWAP '-' =  OR ;           -- test for AddOp
: mulop?   ( char -- tf )  DUP '*' =  SWAP '/' =  OR ;           -- test for MulOp
: emits    ( c-addr u -- ) Tab EMIT TYPE ;                       -- output a string
with tab
: emitdln  ( c-addr u -- ) CR TYPE ;                             -- output a string
without tab, +crlf
: emitln   ( c-addr u -- ) CR emits ;                            -- output a string
with tab and crlf

-- abort, reporting offending item
: name.aborts ( c-addr1 u1 c-addr2 u2 )
        &` CHAR-PREPEND &' CHAR-APPEND $+ aborts ;

: .undefined ( c-addr u -- )
        S" Undefined identifier " 2SWAP name.aborts ;

-- generate a unique label
: newlabel ( -- c-addr u ) S" @" Lcount U>D <# #S #> $+  1 +TO Lcount ;

-- post a label to output
: postlabel ( c-addr u -- ) CR TYPE ':' EMIT ;

-- Recognize a relop
: relop? ( char -- tf )
        DUP  '=' =
        OVER '#' = OR
        OVER '<' = OR
        SWAP '>' = OR ;

: char+!   ( c addr -- )   DUP >S COUNT + C!  1 S> C+! ;

-- skip white space
: skipwhite ( -- )
        BEGIN  Look white?
        WHILE  getchar
        REPEAT ;

-- skip cr+lf
: fin ( -- )
        Look ^M = IF getchar ENDIF
        Look ^J = IF getchar ENDIF
        skipwhite ;

: newline ( -- ) BEGIN  Look ^M = WHILE  fin  REPEAT ;

-- match a specific input character
: match ( char -- )
        DUP Look = IF  DROP getchar skipwhite
                 ELSE  S" `" ROT CHAR-APPEND &' CHAR-APPEND expected
                ENDIF ;

-- get an identifier.
: getname ( -- c-addr u )
        newline
        Look alpha? 0= IF  S" Name" expected  ENDIF
```

```
        token$ C0!
        BEGIN   Look alnum?
        WHILE   Look >UPC token$ char+!  getchar
        REPEAT
        token$ COUNT  skipwhite ;

-- get a number
: getnum ( -- val )
        newline
        Look digit? 0= IF  S" Integer" expected  ENDIF
        0 >R
        BEGIN   Look digit?
        WHILE   R> #10 *  Look '0' - +  >R
                getchar
        REPEAT  '#' TO token
        R>  skipwhite ;

-- Get an identifier and scan it for keywords
: scan ( -- ) getname 0 KWlist lookup KW->token ;

-- Add a new entry to symbol table
: addentry ( c-addr u type -- )
        [cnt]ST LOCALS| #entries T sz addr |
        addr sz intable IF  S" Duplicate identifier " addr sz name.aborts  ENDIF
        #entries @ maxentry = IF  S" Symbol table full" aborts  ENDIF
        1 #entries +!
        addr sz  #entries @ ST  PACK DROP
        T #entries @ SType C! ;

-- Match a specific input string
: matchstring ( c-addr u -- )
        token$ COUNT 2OVER
        COMPARE IF  &` CHAR-PREPEND &' CHAR-APPEND expected
              ELSE  2DROP
             ENDIF ;

-- Code generation
------------------------------------------------------------------------------------
-------------------

-- Load/store primary register with a constant/variable
: loadconstant  ( u -- ) (.) S"  d# -> eax  mov," $+ emitln ;

: loadvariable  ( c-addr u -- )
        2DUP intable 0= IF  .undefined  ENDIF
        S"  dword-ptr -> eax  mov," $+ emitln ;

: storevariable ( c-addr u -- )
        2DUP intable 0= IF  .undefined  ENDIF
        S" eax -> " 2SWAP $+ S"  dword-ptr  mov," $+ emitln ;

-- Allocate storage for a static variable
: allocatestorage ( c-addr1 u1 c-addr2 u2 -- )
        S" CREATE " 2ROT $+ BL CHAR-APPEND 2SWAP $+ S"  , " $+ emitdln ;

: callfunction  ( c-addr u -- ) S"  offset NEAR call," $+ emitln ;              --
call a function

: _negate ( -- ) S" eax  neg,"   emitln ;                                       --
negate primary
: _not    ( -- ) S" eax  not,"   emitln ;                                       --
not primary
: _push   ( -- ) S" eax  push,"  emitln ;                                       --
push primary to stack
: _pop    ( -- ) S" eax  pop,"   emitln ;                                       --
pop primary from stack
```

```
: _decr   ( -- ) S" 1 b# -> eax sub," emitln ;                                 --
decrement primary
: _incr   ( -- ) S" 1 b# -> eax add," emitln ;                                 --
increment primary
: _decSP  ( -- ) S" [esp -4 +] -> esp lea," emitln ;                           --
drop 1 stack item
: _incSP  ( -- ) S" [esp  4 +] -> esp lea," emitln ;                           --
restore stack item
: _popadd ( -- ) S" [esp] -> eax  add,  [esp 4 +] -> esp lea," emitln ;        --
add TOS to primary
: _popsub ( -- ) S" [esp] -> eax  sub,  [esp 4 +] -> esp lea," emitln _negate ; --
subtract TOS from primary
: _popor  ( -- ) S" [esp] -> eax  or,   [esp 4 +] -> esp lea," emitln ;        --
or TOS to primary
: _popxor ( -- ) S" [esp] -> eax  xor,  [esp 4 +] -> esp lea," emitln ;        --
xor TOS to primary
: _popand ( -- ) S" [esp] -> eax  and,  [esp 4 +] -> esp lea," emitln ;        --
xor TOS to primary
: _popcmp ( -- ) S" [esp] -> eax  cmp,  [esp 4 +] -> esp lea," emitln ;        --
compare TOS to primary
: _sete   ( -- ) S" al  sete,   al -> eax  movsx," emitln ;                    --
set primary if equal
: _setne  ( -- ) S" al  setne,  al -> eax  movsx," emitln ;                    --
set primary if NOT equal
: _setg   ( -- ) S" al  setl,   al -> eax  movsx," emitln ;                    --
set primary if greater
: _setl   ( -- ) S" al  setg,   al -> eax  movsx," emitln ;                    --
set primary if less
: _setge  ( -- ) S" al  setle,  al -> eax  movsx," emitln ;                    --
set primary if greater or equal
: _setle  ( -- ) S" al  setge,  al -> eax  movsx," emitln ;                    --
set primary if less or equal

-- jump always; jump if eax 0
: _branch    ( c-addr u -- ) S"  offset NEAR  jmp," $+ emitln ;
: _pcmp+b0>  ( c-addr u -- ) _popcmp S"  offset NEAR  jg,"  $+ emitln ;
: _branch0   ( c-addr u -- ) S" eax -> eax or,  " 2SWAP $+ S"  offset NEAR  je,"  $+
emitln ;
: _branch<>0 ( c-addr u -- ) S" eax -> eax or,  " 2SWAP $+ S"  offset NEAR  jnz," $+
emitln ;

-- multiply TOS and primary; divide primary by TOS
: _popmul ( -- ) S" [esp] dword mul, [esp 4 +] -> esp lea," emitln ;
: _popdiv ( -- ) S" ecx pop, ecx -> eax xchg, eax -> edx mov, #31 b# -> edx sar, ebx
idiv," emitln ;

-- read to primary register and store in variable
: readvar  ( c-addr u -- ) S" READ" callfunction  storevariable ;

-- write from primary register
: writevar ( -- ) S" WRITE" callfunction ;

-- Write assembler header info; epilogue, and prologue
: header ( -- )
        S" -- DATA section --------" emitdln
        S" " emitdln ;

: prolog ( -- )
        S" " emitln
        S" -- CODE section --------" emitdln
        S" "         emitdln
        S" CODE " prog$ COUNT $+ emitdln
        S" rpush,"    emitln
        newlabel postlabel ;

: epilog ( -- )
        S" rpop,  ebx jmp," emitln
```

```
        S" END-CODE"        emitdln ;

-- Expressions
--------------------------------------------------------------------------

-- parse and translate an identifier.
-- Note that we need to store getname locally because parsing a call's
-- parameterlist may clobber it.
: ident ( -- )
        getname DUP 1+ ALLOCATE ?ALLOCATE DUP LOCAL name PACK DROP
        Look '(' = IF   '(' match ')' match
                        name COUNT callfunction
                 ELSE   name COUNT loadvariable
                ENDIF
        name FREE ?ALLOCATE ;

DEFER expression
DEFER boolexpression

-- parse and translate a math factor
: factor ( -- )
        Look '('  = IF  '(' match  boolexpression  ')' match EXIT  ENDIF
        Look alpha? IF  getname loadvariable  EXIT  ENDIF
        getnum loadconstant ;

-- parse and translate the first math factor
: signedfactor ( -- )
        Look '+'  = IF  getchar       ENDIF
        Look '-' <> IF  factor  EXIT  ENDIF
        getchar skipwhite
        Look digit? IF  getnum NEGATE loadconstant
                  ELSE  factor _negate
                 ENDIF ;

-- recognize and translate multiply / divide
: multiply  ( -- ) '*' match  factor _popmul ;
: divide    ( -- ) '/' match  factor _popdiv ;

-- parse and translate a math term
: term ( -- )
        signedfactor
        BEGIN Look mulop?
        WHILE   _push
                CASE Look
                  '*' OF multiply  ENDOF
                  '/' OF divide    ENDOF
                         S" Mulop" expected
                ENDCASE
        REPEAT ;

-- recognize and translate add / subtract
: add       ( -- ) '+' match  term   _popadd ;
: subtract  ( -- ) '-' match  term   _popsub ;

-- parse and translate an expression
:NONAME ( -- )
        term
        BEGIN   Look addop?
        WHILE   _push
                CASE Look
                  '+' OF add      ENDOF
                  '-' OF subtract ENDOF
                         S" Addop" expected
                ENDCASE
        REPEAT ; IS expression

-- recognize and translate a relational "equals"
```

```
: equals      ( -- )  '=' match  expression _popcmp _sete ;

-- recognize and translate a relational "less or equal"
: lessorequal ( -- )  '=' match  expression _popcmp _setle ;

-- recognize and translate a relational "not equals"
: notequals   ( -- )  '>' match  expression _popcmp _setne ;

-- recognize and translate a relational "less than"
: less ( -- )
        '<' match
        Look '=' = IF  lessorequal EXIT  ENDIF
        Look '>' = IF  notequals   EXIT  ENDIF
        expression _popcmp _setl ;

-- recognize and translate a relational "greater than"
: greater ( -- )
        '>' match
        Look '=' = IF  '=' match expression _popcmp _setge EXIT  ENDIF
        expression _popcmp _setg ;

-- parse and translate a relation
: relation ( -- )
        expression
        Look relop? 0= ?EXIT
        _push
        CASE Look
          '=' OF  equals     ENDOF
          '<' OF  less       ENDOF
          '>' OF  greater    ENDOF
        ENDCASE ;

-- parse and translate a boolean factor with NOT
: notfactor ( -- )
        Look '!' <> IF  relation EXIT  ENDIF
        '!' match  relation _not ;

-- parse and translate a boolean term
: boolterm ( -- )
        notfactor
        BEGIN   Look '&' =
        WHILE   _push  '&' match  notfactor _popand
        REPEAT ;

-- Recognize and translate a boolean OR
: boolOR ( -- )  '|' match boolterm _popor ;

-- Recognize and translate an EXCLUSIVE OR
: boolXOR ( -- ) '~' match boolterm _popxor ;

-- Parse and translate a boolean expression
:NONAME ( -- )
        boolterm
        BEGIN   Look orop?
        WHILE   _push
                CASE Look
                  '|' OF  boolOR  ENDOF
                  '~' OF  boolXOR ENDOF
                ENDCASE
        REPEAT ; IS boolexpression

-- Block statements
--------------------------------------------------------------------------------

DEFER block

-- Parse and translate an assignment statement
```

```
: assignment ( -- )
        token$ COUNT DUP 1+ ALLOCATE ?ALLOCATE DUP LOCAL name PACK DROP
        '=' match
        boolexpression
        name COUNT storevariable
        name FREE ?ALLOCATE ;

-- Recognize and translate an IF construct
: doIF ( label -- )
        0 0 LOCALS| L2 L1 L |
        boolexpression
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L1 PACK DROP
        L1 COUNT DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP
        L1 COUNT _branch0
        L block
        token 'l' = IF  L2 FREE ?ALLOCATE
                        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP
                        L2 COUNT _branch
                        L1 COUNT postlabel
                        L block
                 ENDIF
        L2 COUNT postlabel
        S" ENDIF" matchstring

        L1 FREE ?ALLOCATE
        L2 FREE ?ALLOCATE ;

: doWHILE ( -- )
        0 0 LOCALS| L2 L1 |
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L1 PACK DROP
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP

        L1 COUNT postlabel
        boolexpression
        L2 COUNT _branch0
        L2 block
        L1 COUNT _branch
        L2 COUNT postlabel
        S" ENDWHILE" matchstring

        L1 FREE ?ALLOCATE
        L2 FREE ?ALLOCATE ;

-- Parse and translate a loop statement
: doLOOP ( -- )
        0 0 LOCALS| L2 L1 |
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L1 PACK DROP
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP

        L1 COUNT postlabel
        L2 block
        L1 COUNT _branch
        L2 COUNT postlabel
        S" ENDLOOP" matchstring

        L1 FREE ?ALLOCATE
        L2 FREE ?ALLOCATE ;

-- Parse and translate a REPEAT statement
: doREPEAT ( -- )
        0 0 LOCALS| L2 L1 |
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L1 PACK DROP
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP

        L1 COUNT postlabel
        L2 block
        S" UNTIL" matchstring
```

```
        boolexpression
        L1 COUNT _branch0
        L2 COUNT postlabel

        L1 FREE ?ALLOCATE
        L2 FREE ?ALLOCATE ;

-- Parse and translate a for statement
: doFOR ( -- )
        0 0 0 LOCALS| name L2 L1 |
        getname  DUP 1+ ALLOCATE ?ALLOCATE DUP TO name PACK DROP
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L1   PACK DROP
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2   PACK DROP

        '=' match  expression
        _decr
        name COUNT storevariable
        scan S" TO" matchstring  expression
        _push
        L1 COUNT postlabel
        name COUNT loadvariable
        _incr
        name COUNT storevariable
        L2 COUNT _pcmp+b0>
        L2 block
        L1 COUNT _branch
        L2 COUNT postlabel
        S" ENDFOR" matchstring
        _incSP

        name FREE ?ALLOCATE
        L1   FREE ?ALLOCATE
        L2   FREE ?ALLOCATE ;

-- Parse and Translate a DO Statement
: doDO ( -- )
        0 0 LOCALS| L2 L1 |
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L1 PACK DROP
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP

        expression
        L1 COUNT postlabel
        _push
        L2 block
        _pop
        _decr
        L1 COUNT _branch<>0
        _decSP
        L2 COUNT postlabel
        S" ENDDO" matchstring
        _incSP

        L1 FREE ?ALLOCATE
        L2 FREE ?ALLOCATE ;

-- Recognize and translate a BREAK
: doBREAK ( label -- )
        DUP C@ 0= IF  S" No loop to break from" aborts  ENDIF
        COUNT _branch ;

-- Process a read statement
: doread ( -- )
        '(' match
        getname readvar
        BEGIN  Look ',' =
        WHILE  ',' match getname readvar
        REPEAT
```

```
        ')' match ;

-- Process a write statement
: dowrite ( -- )
        '(' match
        expression writevar
        BEGIN  Look ',' =
        WHILE  ',' match expression writevar
        REPEAT
        ')' match ;

-- Recognize and translate a statement block
:NONAME ( label -- )
        LOCAL L
        scan
        BEGIN  token 'e' <>
               token 'l' <> AND
               token 'u' <> AND
        WHILE  CASE token
                'i' OF  L doIF    ENDOF
                'w' OF  doWHILE   ENDOF
                'd' OF  doDO      ENDOF
                'L' OF  doLOOP    ENDOF
                'r' OF  doREPEAT  ENDOF
                'f' OF  doFOR     ENDOF
                'B' OF  L doBREAK ENDOF
                'R' OF  doread    ENDOF
                'W' OF  dowrite   ENDOF
                        assignment
               ENDCASE
               scan
        REPEAT ; IS block

-- Declarations
------------------------------------------------------------------------------------

-- Allocate storage for a variable
: alloc ( c-addr u -- )
        LOCALS| sz addr |
        addr sz intable IF  S" Duplicate variable name " addr sz name.aborts  ENDIF
        addr sz 'v' addentry
        addr sz
        Look '=' = IF   '=' match
                        Look '-' = IF   '-' match
                                        S" -"
                                 ELSE   S" "
                                ENDIF
                        getnum (.) $+
                 ELSE   S" 0"
                ENDIF   allocatestorage ;

-- Parse and translate a data declaration
: decl ( -- )
        getname alloc
        BEGIN  Look ',' =
        WHILE  ',' match  getname alloc
        REPEAT ;

-- Parse and translate the global declarations
: topdecls ( -- )
        scan
        BEGIN   token 'b' <>
        WHILE   CASE token
                 'v' OF decl  ENDOF
                     S" Unrecognized keyword " token$ COUNT name.aborts
                ENDCASE
                scan
```

```
        REPEAT ;

-- Parse and translate MAIN code
: doprogram ( -- )
        S" BEGIN" matchstring
        prolog
        C" " block
        S" END" matchstring
        epilog ;

-- Parse and translate a program
--------------------------------------------------------------------------
: prog ( -- )
        S" PROGRAM" matchstring
        getname prog$ PACK DROP
        header
        topdecls
        doprogram
        '.' match ;

-- initialize
: init ( -- )
        CLEAR Lcount
        [cnt]ST    OFF
        [cnt]SType OFF
        CR getchar scan ;

: TINY10 ( -- )
        init prog
        Look ^M <> IF  S" Unexpected data after " S" ." name.aborts  ENDIF ;
```

11. Пересмотр лексического анализа

ВВЕДЕНИЕ

У меня есть хорошие и плохие новости. Плохие новости - эта глава не та, которую я вам обещал последний раз. Более того, и следующая глава также.

Хорошие новости в причине появления этой главы: я нашел способ упростить и улучшить лексический анализатор компилятора. Позвольте мне объяснить.

ПРЕДПОСЫЛКА

Если вы помните, мы подробно говорили на тему лексических анализаторов в Главе 7 и я оставил вас с проектом распределенного сканера который, я чувствовал, был почти настолько простым, насколько я смог сделать... более чем большинство из того, что я где-либо видел. Мы использовали эту идею в Главе 10. Полученная структура компилятора была простой и она делала свою работу.

Однако недавно я начал испытывать проблемы такого рода, которые подсказывают, что возможно вы делаете что-то неправильно.

Проблемы достигли критической стадии когда я попытался обратиться к вопросу точек с запятой. Некоторые люди спрашивали меня, действительно ли KISS будет использовать их для разделения операторов. Я не намеревался использовать точки с запятой просто потому, что они мне не нравятся и, как вы можете видеть, они не доказали своей необходимости.

Но я знаю, что многие из вас, как и я, привыкли к ним, так что я намеревался написать короткую главу чтобы показать вам, как легко они могут быть добавлены раз вы так склонны к ним.

Чтож, оказалось что их совсем непросто добавить. Фактически это было чрезвычайно трудно.

Я полагаю, что должен был понять что что-то было неправильно из-за проблемы переносов строк. В последних двух главах мы обращались к этому вопросу и я показал вам, как работать с переносами с помощью процедуры, названной достаточно соответствующе NewLine. В TINY Version 1.0 я расставил вызовы этой процедуры в стратегических местах кода.

Кажется, что всякий раз, когда я обращался к проблеме переносов, я, однако, находил этот вопрос сложным и полученный синтаксически анализатор оказывался очень ненадежным... одно удаление или добавление здесь или там и все разваливалось. Оглядываясь назад, я понимаю, что это было предупреждение, на которое я просто не обращал внимания.

Когда я попробовал добавить точку с запятой к переносам строк это стало последней каплей. Я получил слишком сложное решение. Я начал понимать, что необходимо что-то менять коренным образом.

Итак, в некотором отношении эта глава заставить нас возвратиться немного назад и пересмотреть заново вопрос лексического анализа. Сожалею об этом. Это цена, которую вы платите за возможность следить за мной в режиме реального времени. Но новая версия определенно усовершенствована и хорошо послужит нам дальше.

Как я сказал, сканер использованный нами в Главе 10, был почти настолько простым, насколько возможно. Но все может быть улучшено. Новый сканер более похож на классический сканер и не так прост как прежде. Но общая структура компилятора даже проще чем раньше. Она также более надежная и проще для добавления и/или модификации. Я думаю, что она стоит времени, потраченного на это отклонение. Так что в этой главе я покажу вам новую структуру. Без сомнения вы будете счастливы узнать, что хотя изменения влияют на многие процедуры, они не очень глубоки и поэтому мы теряем не очень многое из того что было сделано до этого.

Как ни странно, новый сканер намного более стандартен чем старый и он очень похож на более общий сканер, показанный мной ранее в главе 7. Вы должны помнить день, когда я сказал: K-I-S-S!

ПРОБЛЕМА

Проблема начинает проявлять себя в процедуре Block, которую я воспроизвел ниже:

```
-- ----------------------------------------------------------------
-- Parse and translate a block of statements
:NONAME ( -- )
        scan
        BEGIN  token 'e' <>
               token 'l' <> AND
        WHILE  CASE token
                'i' OF  doIF      ENDOF
                'w' OF  doWHILE   ENDOF
                'R' OF  doread    ENDOF
                'W' OF  dowrite   ENDOF
                        assignment
               ENDCASE
               scan
        REPEAT ; IS block
-- ----------------------------------------------------------------
```

Как вы можете видеть, Block ориентирован на индивидуальные утверждения программы. При каждом проходе цикла мы знаем, что мы находимся в начале утверждения. Мы выходим из блока когда обнаруживаем END или ELSE.

Но предположим, что вместо этого мы встретили точку с запятой. Вышеуказанная процедура не может ее обработать, так как процедура Scan ожидает и может принимать только токены, начинающиеся с буквы.

Я повозился с этим немного чтобы найти исправление. Я нашел множество возможных подходов, но ни один из них меня не удовлетворял. В конце концов я выяснил причину.

Вспомните, что когда мы начинали с наших односимвольных синтаксических анализаторов, мы приняли соглашение, по которому предсказывающий символ должен быть всегда предварительно считан. То есть, мы имели бы символ, соответствующий нашей текущей позиции во входном потоке, помещенный в глобальной символьной переменной Look, так что мы могли проверять его столько раз, сколько необходимо. По правилу, которое мы приняли, каждый распознаватель, если он находил предназначенный ему символ, перемещал бы Look на следующий символ во входном потоке.

Это простое и фиксированное соглашение служило нам очень хорошо когда мы имели односимвольные токены, и все еще служит. Был бы большой смысл применить то же самое правило и к многосимвольным токенам.

Но когда мы залезли в лексический анализ, я начал нарушать это простое правило. Сканер из Главы 10 действительно продвигался к следующему токену если он находил идентификатор или ключевое слово, но он не делал этого если находил возврат каретки, символ пробела или оператор.

Теперь, такой смешанный режим работы ввергает нас в глубокую проблему в процедуре Block, потому что был или нет входной поток продвинут зависит от вида встреченного нами токена. Если это ключевое слово или левая часть операции присваивания, "курсор", как определено содержимым Look, был продвинут к следующему символу или к началу незаполненного пространства. Если, с другой стороны, токен является точкой с запятой, или если мы нажали возврат каретки курсор не был продвинут.

Само собой разумеется, мы можем добавить достаточно логики чтобы удержаться на правильном пути. Но это сложно и делает весь анализатор очень ненадежным.

Существует гораздо лучший способ - просто принять то же самое правило, которое так хорошо работало раньше, и относиться к токенам так же как одиночным сиволам. Другими словами, мы будем заранее считывать токен подобно тому, как мы всегда считывали символ. Это кажется таким очевидным как только вы подумаете об этом способе.

Достаточно интересно, что если мы поступим таким образом, существующая проблема с символами перевода строки исчезнет. Мы можем просто рассматривать их как символы пробела, таким образом обработка переносов становится тривиальной и значительно менее склонной к ошибкам чем раньше.

РЕШЕНИЕ

Давайте начнем решение проблемы с пересмотра двух процедуры:

```
-- -------------------------------------------------------------
-- Get an identifier
: getname ( -- )
        skipwhite
        Look alpha? 0= IF  S" Identifier" expected  ENDIF
        'x' TO token
        token$ C0!
        BEGIN
          Look >UPC token$ char+!
          getchar
          Look alnum? 0=
        UNTIL ;
-- -------------------------------------------------------------
-- Get a number
: getnum ( -- )
        skipwhite
        Look digit? 0= IF  S" Number" expected  ENDIF
        '#' TO token
        token$ C0!
        BEGIN
```

```
          Look token$ char+!
          getchar
          Look digit? 0=
        UNTIL ;
-- --------------------------------------------------------------
```

Эти две процедуры функционально почти идентичны тем, которые я показал вам в Главе 7. Каждая из них выбирает текущий токен, или идентификатор или число, в глобальную строковую переменную Value. Они также присваивают кодированной версии, Token, соответствующий код. Входной поток останавливается на Look, содержащем первый символ, не являющийся частью токена.

Мы можем сделать то же самое для операторов, даже многосимвольных, с помощью процедуры типа:

```
-- --------------------------------------------------------------
-- Get an operator
: getop ( -- )
        Look TO token
        token$ C0!
        BEGIN
          Look token$ char+!
          getchar
          Look digit?  Look alpha? OR  Look white? OR
        UNTIL ;
-- --------------------------------------------------------------
```

Обратите внимание, что GetOps возвращает в качестве закодированного токена первый символ оператора. Это важно, потому что это означает, что теперь мы можем использовать этот одиночный символ для управления синтаксическим анализатором вместо предсказывающего символа.

Нам нужно связать эти процедуры вместе в одну процедуру, которая может обрабатывать все три случая. Следующая процедура будет считывать любой из этих типов токенов и всегда оставлять входной поток за ним:

```
-- --------------------------------------------------------------
-- Get the next input token
: next ( -- )
        skipwhite
        Look alpha? IF  getname EXIT  ENDIF
        Look digit? IF  getnum  EXIT  ENDIF
        getop ;
-- --------------------------------------------------------------
```

Обратите внимание, что здесь я поместил SkipWhite перед вызовами а не после. Это означает в основном, что переменная Look не будет содержать значимого значения и, следовательно, мы не должны использовать ее как тестируемое значение при синтаксическом анализе, как мы делали до этого. Это большое отклонение от нашего нормального подхода.

Теперь, не забудьте, что раньше я избегал обработки символов возврата каретки (CR) и перевода строки (LF) как незаполненного пространства. Причина была в том, что так как SkipWhite вызывается последней в сканере, встреча с LF инициировала бы чтение из входного потока. Если бы мы были на последней строке программы, мы не могли бы выйти до тех пор, пока мы не введем другую строку с отличным от пробела символом. Именно поэтому мне требовалась вторая процедура NewLine для обработки CRLF.

Но сейчас, когда первым происходит вызов SkipWhite, это то поведение, которое нам нужно. Компилятор должен знать, что появился другой токен или он не должен вызывать Next. Другими словами, он еще не обнаружил завершающий END. Поэтому мы будем настаивать на дополнительных данных до тех пор, пока не найдем что-либо.

Все это означает, что мы можем значительно упростить и программу и концепции, обрабатывая CR и LF как незаполненное простанство и убрав NewLine. Вы можете сделать это просто изменив функцию IsWhite:

```
-- ----------------------------------------------------------------
-- Recognize white space
: white? ( char -- tf )
        DUP  BL  =
        OVER Tab = OR
        OVER ^M  = OR
        SWAP ^J  = OR ;
-- ----------------------------------------------------------------
```

Мы уже пробовали аналогичные подпрограммы в Главе 7, но вы могли бы также попробовать и эти. Добавьте их к копии Cradle и вызовите Next в основной программе:

```
-- ----------------------------------------------------------------
-- main program
        init
        BEGIN
          next
          CR Token EMIT SPACE token$ .$
          Token '.' =
        UNTIL ;
-- ----------------------------------------------------------------
```

Откомпилируйте и проверьте, что вы можете разделять программу на серии токенов и вы получаете правильные кода для каждого токена.

Почти работает, но не совсем. Существуют две потенциальные проблемы: Во-первых, в KISS/TINY почти все наши операторы - односимвольные. Единственное исключение составляют операторы отношений >=, <= и <>. Было бы позором обрабатывать все операторы как строки и выполнять сравнение строк когда почти всегда удовлетворит сравнение одиночных символов. Второе, и более важное, программа не работает, когда два оператора появляются вместе как в (a+b)*(c+d). Здесь строка после b была бы интерпретирована как один оператор ")*(".

Можно устранить эту проблему. К примеру мы могли бы просто дать GetOp список допустимых символов и обрабатывать скобки как отличный от других тип операторов. Но это хлопотное дело.

К счастью, имеется лучший способ, который решает все эти проблемы. Так как почти все операторы односимвольные, давайте просто позволим GetOp получать только один символ одновременно. Это не только упрощает GetOp, но также немного ускоряет программу. У нас все еще остается проблема операторов отношений, но мы в любом случае обрабатывали их как специальные случаи.

Так что вот финальная версия GetOp:

```
-- ----------------------------------------------------------------
-- Get an operator
: getop ( -- )
        skipwhite
        Look TO Token
        Look token$ char-place
        getchar ;
-- ----------------------------------------------------------------
```

Обратите внимание, что я все еще присваиваю Value значение. Если вас действительно затрагивает эффективность, вы могли бы это опустить. Когда мы ожидаем оператор, мы в любом случае будем проверять только Token, так что значение этой строки не будет иметь значение. Но мне кажется хорошая практика дать ей значение на всякий случай.

Испытайте эту версию с каким-нибудь реалистично выглядящим кодом. Вы должны быть способны разделять любую программу на ее индивидуальные токены, но предупреждаю, что двухсимвольные операторы отношений будут отсканированы как два раздельных токена. Это нормально... мы будем выполнять их синтаксический анализ таким способом.

Теперь, в главе 7 функция Next была объединена с процедурой Scan, которая также сверяла каждый идентификатор со списком ключевых слов и кодировала каждый найденный. Как я упомянул тогда, последнее, что мы захотели бы сделать - использовать такую процедуру в местах, где ключевые слова не должны появляться, таких как выражения. Если бы мы сделали это, список ключевых слов просматривался бы для каждого идентификатора, появляющегося в коде. Нехорошо.

Правильней было бы в этом случае просто разделить функции выборки токенов и поиска ключевых слов. Версия Scan, показанная ниже, только проверяет ключевые слова. Обратите внимание, что она оперирует текущим токеном и не продвигает входной поток.

```
-- ------------------------------------------------------------
-- Scan the current identifier for keywords
: scan ( -- )
        Token 'x' = IF  token$ COUNT  0 KWlist lookup
                        KW->token
                 ENDIF ;
-- ------------------------------------------------------------
```

Последняя деталь. В компиляторе есть несколько мест, в которых мы должны фактически проверить строковое значение токена. В основном это сделано для того, чтобы различать разные END, но есть и пара других мест. (Я должен заметить, между прочим, что мы могли бы навсегда устранить потребность в сравнении символов END кодируя каждый из них различными символами. Прямо сейчас мы определенно идем маршрутом ленивого человека.)

Следующая версия MatchString замещает символьно-ориентированную Match. Заметьте, что как и Match она не продвигает входной поток.

```
-- ------------------------------------------------------------
-- Match a specific input string
: matchstring ( c-addr u -- )
        token$ COUNT 2OVER
        COMPARE IF  &` CHAR-PREPEND &' CHAR-APPEND expected
              ELSE  2DROP
             ENDIF
        next ;
-- ------------------------------------------------------------
```

ИСПРАВЛЕНИЕ КОМПИЛЯТОРА

Вооружившись этими новыми процедурами лексического анализа мы можем теперь начать исправлять компилятор. Изменения весьма незначительные, но есть довольно много мест, где они необходимы. Вместо того, чтобы показывать вам каждое место я дам вам общую идею а затем просто покажу готовый продукт.

Прежде всего, код процедуры Block не изменяется, но меняется ее назначение:

```
-- ------------------------------------------------------------
-- Parse and translate a block of statements
:NONAME ( -- )
        scan
        BEGIN  token 'e' <>
               token 'l' <> AND
        WHILE  CASE token
                'i' OF  doIF      ENDOF
                'w' OF  doWHILE   ENDOF
                'R' OF  doread    ENDOF
```

```
                'W' OF  dowrite   ENDOF
                        assignment
               ENDCASE
               scan
        REPEAT ; IS block
-- -----------------------------------------------------------
```

Не забудьте, что новая версия Scan не продвигает входной поток, она только сканирует ключевые слова. Входной поток должен продвигаться каждой процедурой, которую вызывает Block.

В общих чертах, мы должны заменить каждую проверку Look на аналогичную проверку Token. Например:

```
-- ------------------------------------------------------------
-- Parse and translate a boolean expression
:NONAME ( -- )
        boolterm
        BEGIN   Token orop?
        WHILE   _push
                CASE Token
                  '|' OF  boolOR  ENDOF
                  '~' OF  boolXOR ENDOF
                ENDCASE
        REPEAT ; IS boolexpression
-- -----------------------------------------------------------
```

В процедурах типа Add мы больше не должны использовать Match. Нам необходимо только вызывать Next для продвижения входного потока:

```
-- -----------------------------------------------------------
-- Recognize and translate an add
: add       ( -- ) next term   _popadd ;
-- -----------------------------------------------------------
```

Управляющие структуры фактически более простые. Мы просто вызываем Next для продвижения через ключевые слова управляющих конструкций:

```
-- -----------------------------------------------------------
-- Recognize and translate an if construct

DEFER block

: doIF ( -- )
        0 0 LOCALS| L2 L1 |
        next
        boolexpression
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L1 PACK DROP
        L1 COUNT DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP
        L1 COUNT _branch0
        block
        token 'l' = IF  L2 FREE ?ALLOCATE
                        next
                        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP
                        L2 COUNT _branch
                        L1 COUNT postlabel
                        block
                ENDIF
        L2 COUNT postlabel
        S" ENDIF" matchstring

        L1 FREE ?ALLOCATE
        L2 FREE ?ALLOCATE ;
```

```
-- --------------------------------------------------------------
```

Это все необходимые изменения. В листинге Tiny Version 1.1, данном ниже, я также сделал ряд других "усовершенствований", которые в действительности не нужны. Позвольте мне кратко разъяснить их:

1. Я удалил две процедуры Prog и Main и объединил их функции в основной программе. Они кажется не добавляли ясности... фактически они просто немного загрязняли программу.
2. Я удалил ключевые слова PROGRAM и BEGIN из списка ключевых слов. Каждое из них появляется в одном месте, так что нет необходимости искать его.
3. Обжегшись однажды на чрезмерной дозе сообразительности, я напомнил себе, что TINY предназначен быть минималистским языком. Поэтому я заменил причудливую обработку унарного минуса на самую простую какую мог придумать. Гигантский шаг назад в качестве кода, но огромное упрощение компилятора. Для использования другой версии правильным местом был бы KISS.
4. Я добавил несколько подпрограмм проверок ошибок типа CheckTable и CheckDup и заменил встроенный код на их вызовы. Это навело порядок во многих подпрограммах.
5. Я убрал проверку ошибок из подпрограмм генерации кода типа Store и поместил их в подпрограммы анализа, к которым они относятся. Смотрите например Assignment.
6. Существовала ошибка в InTable и Locate которая заставляла их проверять все позиции вместо позиций только с достоверными данными. Теперь они проверяют только допустимые ячейки. Это позволяет нам устранить необходимость инициализации таблицы идентификаторов, которая была в Init.
7. Процедура AddEntry теперь имеет два параметра, что помогает сделать программу немного более модульной.
8. Я подчистил код для операторов отношений добавив новые процедуры CompareExpression и NextExpression.
9. Я устранил ошибку в подпрограмме Read... старая версия не выполняла проверку на правильность имени переменной.

ЗАКЛЮЧЕНИЕ

Полученный компилятор Tiny показан ниж(tiny11.frt).е. Не считая удаленного ключевого слова PROGRAM он анализирует тот же самый язык что и раньше. Он просто немного чище и, что более важно, значительно более надежный. Он мне нравится.

В следующей главе будет другое отклонение: сперва обсуждение точек с запятой и все, что привело меня такому беспорядку. Затем мы займемся процедурами и типами. Добавление этих возможностей далеко продвинет нас на пути к выведению KISS из категории "игрушечных языков". Мы подобрались очень близко к возможности написать серьезный компилятор.

```
-- --------------------------------------------------------------
-- Variable declarations

-1 =: unknown

0 VALUE Lcount                          -- label counter
0 VALUE Look                            -- lookahead character
0 VALUE token                           -- encoded token

#100 =: maxentry

CREATE token$  0 C, #256 CHARS ALLOT    -- unencoded token
CREATE prog$   0 C, #256 CHARS ALLOT    -- program name

: char+!     ( c addr -- ) DUP >S COUNT + C!  1 S> C+! ;
: char-place ( c addr -- ) 1 OVER C!  CHAR+ C! ;
```

```
:   token> ( -- c-addr u ) token$ COUNT ;
:  =token  ( char -- tf )  token =  ;
: <>token  ( char -- tf )  token <> ;

-- Symbol tables and labels -----------------------------------

-- Helper to build symbol tables
: $, ( c-addr u size -- )
        >S S MIN >R
        R@ C, R@ 0 ?DO  C@+ C,  LOOP DROP
        S> R> - CHARS ALLOT ;

-- Type declarations
: SYMTAB ( size -- )
        CREATE  DUP 0< IF  0 , ABS DUP ,
                           1+ maxentry * ALLOT
                     ELSE  HERE >R
                           0 ,  ( #items )
                           DUP >S ,  ( itemsize )
                           BEGIN  BL <WORD> DUP
                           WHILE  2DUP S" \" COMPARE
                                  0= IF  2DROP REFILL DROP
                                   ELSE  S $, 1 R@ +!
                                  ENDIF
                           REPEAT 2DROP -R -S
                    ENDIF
        DOES>   CELL+ @+ 1+ ROT * + ( ix -- addr ) ;

-- Definition of keywords and token types

8 CHARS =: /symbol

/symbol SYMTAB KWlist   IF ELSE ENDIF   \
                        WHILE ENDWHILE  \
                        DO ENDDO        \
                        LOOP ENDLOOP    \
                        REPEAT UNTIL    \
                        FOR TO ENDFOR   \
                        BREAK           \
                        READ WRITE      \
                        VAR END

: KW->token ( index -- ) 2+ C" xilewedeLerufteBRWve" + C@ TO token ;

        /symbol  NEGATE SYMTAB ST
        =CELL 1- NEGATE SYMTAB SType

0 ST    2 CELLS - =: [cnt]ST
0 SType 2 CELLS - =: [cnt]SType

: lookup ( c-addr u 'table -- n2|unknown )
        0 0 LOCALS| /symbol n table sz addr |
        table 2 CELLS - @+ TO n  @ TO /symbol
        n 0<= IF  unknown EXIT  ENDIF
           0 n DO
                 /symbol 1+ I * table + COUNT
                 addr sz COMPARE 0= IF  I UNLOOP EXIT  ENDIF
         -1 +LOOP
        unknown ;

-- Locate symbol in table
: locate  ( c-addr u -- ix ) 0 ST lookup ;      -- Returns -1 | the index of the
entry.
: intable ( c-addr u -- tf ) 0 ST lookup 0>= ;  -- Look for symbol in table

-- Dump symbol table
```

```
: .symbols ( -- )
        [cnt]ST @ 0= IF  CR ." No symbols defined." EXIT  ENDIF
        CR ." -- type --.--- name ---"
        [cnt]ST @ 0 ?DO  CR     5 HTAB I 1+ SType C@ EMIT
                              #16 HTAB I 1+ ST    .$
                   LOOP ;

-- Tools ---------------------------------------------------------

: getchar  ( -- )          EKEY TO Look ;                        -- read new
character from input stream
: error    ( c-addr u -- ) CR ^G EMIT ." Error: " TYPE ." ." ;  -- report an error
: aborts   ( c-addr u -- ) error ABORT ;                         -- report error and
halt
: expected ( c-addr u -- ) S"  expected" $+ aborts ;             -- report what was
expected
: alpha?   ( char -- tf )  >UPC 'A' 'Z' 1+ WITHIN ;              -- recognize an
alpha character
: digit?   ( char -- tf )  '0' '9' 1+ WITHIN ;                   -- recognize a
decimal digit
: alnum?   ( char -- tf )  DUP alpha? SWAP digit? OR ;           -- recognize
alphanumeric
: orop?    ( char -- tf )  DUP '|' =  SWAP '~' =  OR ;           -- recognize an OR
operand
: addop?   ( char -- tf )  DUP '+' =  SWAP '-' =  OR ;           -- test for AddOp
: mulop?   ( char -- tf )  DUP '*' =  SWAP '/' =  OR ;           -- test for MulOp
: emits    ( c-addr u -- ) Tab EMIT TYPE ;                       -- output a string
with tab
: emitdln  ( c-addr u -- ) CR TYPE ;                             -- output a string
without tab, +crlf
: emitln   ( c-addr u -- ) CR emits ;                            -- output a string
with tab and crlf

-- Recognize white space
: white? ( char -- tf )
        DUP  Tab =
        OVER BL  = OR
        OVER ^M  = OR
        SWAP ^J  = OR ;

-- Skip white space
: skipwhite ( -- ) BEGIN  Look white?  WHILE  getchar  REPEAT ;

-- Abort, reporting an offending item
: name.aborts ( c-addr1 u1 c-addr2 u2 ) &' CHAR-APPEND S"  `" 2SWAP $+ $+ aborts ;

: .undefined ( c-addr u -- ) S" Undefined identifier" 2SWAP name.aborts ;
: .duplicate ( c-addr u -- ) S" Duplicate identifier" 2SWAP name.aborts ;
: checktable ( c-addr u -- ) 2DUP intable    IF 2DROP EXIT ENDIF .undefined ; -- Is
identifier in the symbol table?
: checkdup   ( c-addr u -- ) 2DUP intable 0= IF 2DROP EXIT ENDIF .duplicate ; -- Is
identifier already in symbol table?
: checkident ( -- )          'x' <>token IF  S" Identifier" expected  ENDIF ; -- Is
current token an identifier?

-- Generate a unique label / post it to output
: newlabel  ( -- c-addr u ) S" @" Lcount U>D <# # # # #> $+  1 +TO Lcount ;
: postlabel ( c-addr u -- ) CR TYPE ':' EMIT ;

-- Recognize a relop
: relop? ( char -- tf )
        DUP  '=' =
        OVER '#' = OR
        OVER '<' = OR
        SWAP '>' = OR ;

-- Match a specific input character
```

```
: match ( char -- )
        DUP Look = IF  DROP getchar skipwhite
                 ELSE  S" `" ROT CHAR-APPEND &' CHAR-APPEND expected
                ENDIF ;

-- Get an identifier
: getname ( -- )
        skipwhite
        Look alpha? 0= IF  S" Identifier" expected  ENDIF
        'x' TO token
        token$ C0!
        BEGIN
          Look >UPC token$ char+!  getchar
          Look alnum? 0=
        UNTIL ;

-- Get a number
: getnum ( -- )
        skipwhite
        Look digit? 0= IF  S" Number" expected  ENDIF
        '#' TO token
        token$ C0!
        BEGIN
          Look token$ char+!
          getchar
          Look digit? 0=
        UNTIL ;

-- Get an operator
: getop ( -- )
        skipwhite
        Look TO token
        Look token$ char-place
        getchar ;

-- Get next token of any type
: next ( -- )
        skipwhite
        Look alpha? IF  getname EXIT  ENDIF
        Look digit? IF  getnum  EXIT  ENDIF
        getop ;

-- Get an identifier and scan it for keywords
: scan ( -- ) 'x' =token IF  token>  0 KWlist lookup  KW->token  ENDIF ;

-- Match a specific input string
: matchstring ( c-addr u -- )
        token> 2OVER
        COMPARE IF  &` CHAR-PREPEND &' CHAR-APPEND expected
              ELSE  2DROP  next
             ENDIF ;

-- Add a new entry to symbol table
: addentry ( c-addr u type -- )
        [cnt]ST LOCALS| #entries T sz addr |
        addr sz checkdup
        #entries @ maxentry = IF  S" Symbol table full" aborts  ENDIF
        1 #entries +!
        addr sz  #entries @ ST     PACK DROP
        T        #entries @ SType  C! ;

-- Code generation ------------------------------------------

-- Load primary register with a constant
: loadconstant  ( c-addr u -- ) S"  d# -> eax  mov," $+ emitln ;

-- Load primary register from variable
```

```
: loadvariable  ( c-addr u -- )
        2DUP intable 0= IF  .undefined  ENDIF
        S"  dword-ptr -> eax  mov," $+ emitln ;

-- Store primary register in variable
: storevariable ( c-addr u -- )
        2DUP intable 0= IF  .undefined  ENDIF
        S" eax -> " 2SWAP $+ S"  dword-ptr  mov," $+ emitln ;

-- Allocate storage for a static variable
: allocatestorage ( $value $name -- )
        S" CREATE " 2SWAP $+ BL CHAR-APPEND 2SWAP $+ S"  , " $+ emitdln ;

: _callw  ( c-addr u -- ) S"  offset NEAR call," $+ emitln ;                    --
call a word
: _clear  ( -- ) S" eax -> eax xor," emitln ;                                   --
clear primary
: _negate ( -- ) S" eax  neg,"   emitln ;                                       --
negate primary
: _not    ( -- ) S" eax  not,"   emitln ;                                       --
not primary
: _push   ( -- ) S" eax  push,"  emitln ;                                       --
push primary to stack
: _pop    ( -- ) S" eax  pop,"   emitln ;                                       --
pop primary from stack
: _decr   ( -- ) S" 1 b# -> eax sub," emitln ;                                  --
decrement primary
: _incr   ( -- ) S" 1 b# -> eax add," emitln ;                                  --
increment primary
: _decSP  ( -- ) S" [esp -4 +] -> esp lea," emitln ;                            --
drop 1 stack item
: _incSP  ( -- ) S" [esp  4 +] -> esp lea," emitln ;                            --
restore stack item
: _popadd ( -- ) S" [esp] -> eax  add,  [esp 4 +] -> esp lea," emitln ;         --
add TOS to primary
: _popsub ( -- ) S" [esp] -> eax  sub,  [esp 4 +] -> esp lea," emitln _negate ; --
subtract TOS from primary
: _popor  ( -- ) S" [esp] -> eax  or,   [esp 4 +] -> esp lea," emitln ;         --
or TOS to primary
: _popxor ( -- ) S" [esp] -> eax  xor,  [esp 4 +] -> esp lea," emitln ;         --
xor TOS to primary
: _popand ( -- ) S" [esp] -> eax  and,  [esp 4 +] -> esp lea," emitln ;         --
xor TOS to primary
: _popcmp ( -- ) S" [esp] -> eax  cmp,  [esp 4 +] -> esp lea," emitln ;         --
compare TOS to primary
: _sete   ( -- ) S" al  sete,   al -> eax  movsx," emitln ;                     --
set primary if equal
: _setne  ( -- ) S" al  setne,  al -> eax  movsx," emitln ;                     --
set primary if NOT equal
: _setg   ( -- ) S" al  setl,   al -> eax  movsx," emitln ;                     --
set primary if greater
: _setl   ( -- ) S" al  setg,   al -> eax  movsx," emitln ;                     --
set primary if less
: _setge  ( -- ) S" al  setle,  al -> eax  movsx," emitln ;                     --
set primary if greater or equal
: _setle  ( -- ) S" al  setge,  al -> eax  movsx," emitln ;                     --
set primary if less or equal

-- A collection of jumps
: _branch    ( c-addr u -- ) S"  offset NEAR  jmp," $+ emitln ;
: _pcmp+b0>  ( c-addr u -- ) _popcmp S"  offset NEAR  jg,"  $+ emitln ;
: _branch0   ( c-addr u -- ) S" eax -> eax or,  " 2SWAP $+ S"  offset NEAR  je,"  $+
emitln ;
: _branch<>0 ( c-addr u -- ) S" eax -> eax or,  " 2SWAP $+ S"  offset NEAR  jnz," $+
emitln ;

-- Multiply TOS and primary; divide primary by TOS
```

```
: _popmul ( -- ) S" [esp] dword mul, [esp 4 +] -> esp lea," emitln ;
: _popdiv ( -- ) S" ecx pop, ecx -> eax xchg, eax -> edx mov, #31 b# -> edx sar, ebx
idiv," emitln ;

-- Read to primary register and store in variable
: readit  ( c-addr u -- ) S" READ" _callw  storevariable ;

-- Write from primary register
: writeit ( -- ) S" WRITE" _callw ;

-- Write assembler header info
: header ( -- )
        S" -- DATA section --------" emitdln
        S" " emitdln ;

-- Write assembler prologue
: prolog ( -- )
        S" " emitln
        S" -- CODE section --------" emitdln
        S" "          emitdln
        S" CODE " prog$ COUNT $+ emitdln
        S" rpush,"    emitln
        newlabel postlabel ;

-- Write assembler epilogue
: epilog ( -- )
        S" rpop,  ebx jmp," emitln
        S" END-CODE"        emitdln ;

-- Expressions ------------------------------------------------

DEFER boolexpression

-- Parse and translate a math factor
: factor ( -- )
        '(' =token IF  next boolexpression  S" )" matchstring  EXIT  ENDIF
        'x' =token IF  token> loadvariable  next  EXIT  ENDIF
        '#' =token IF  token> loadconstant  next  EXIT  ENDIF
        S" Math factor" expected ;

-- Recognize and translate multiply / divide
: multiply  ( -- ) next  factor _popmul ;
: divide    ( -- ) next  factor _popdiv ;

-- Parse and translate a math term
: term ( -- )
        factor
        BEGIN   token mulop?
        WHILE   _push
                CASE token
                  '*' OF multiply  ENDOF
                  '/' OF divide    ENDOF
                ENDCASE
        REPEAT ;

-- Recognize and translate add / subtract
: add       ( -- ) next  term   _popadd ;
: subtract  ( -- ) next  term   _popsub ;

-- Parse and translate an expression
: expression ( -- )
        token addop? IF  _clear
                   ELSE  term
                  ENDIF
        BEGIN   token addop?
        WHILE   _push
                CASE token
```

```
                '+' OF add      ENDOF
                '-' OF subtract ENDOF
              ENDCASE
        REPEAT ;

-- Get another / next expression and compare
: comparex ( -- ) expression _popcmp ;
: nextx    ( -- ) next comparex ;

-- Recognize and translate a relational "equals" / "less or equal" / "not equals"
: equals      ( -- )  nextx _sete  ;
: lessorequal ( -- )  nextx _setle ;
: notequals   ( -- )  nextx _setne ;

-- Recognize and translate a relational "less than"
: less ( -- )
        next
        CASE token
          '=' OF  lessorequal    ENDOF
          '>' OF  notequals      ENDOF
                  comparex _setl
        ENDCASE ;

-- Recognize and translate a relational "greater than"
: greater ( -- )
        next
        '=' =token IF  nextx _setge EXIT  ENDIF
        comparex _setg ;

-- Parse and translate a relation
: relation ( -- )
        expression
        token relop? 0= ?EXIT
        _push
        CASE token
          '=' OF  equals     ENDOF
          '<' OF  less       ENDOF
          '>' OF  greater    ENDOF
        ENDCASE ;

-- Parse and translate a boolean factor with NOT
: notfactor ( -- )
        '!' <>token IF  relation EXIT  ENDIF
        next  relation _not ;

-- Parse and translate a boolean term
: boolterm ( -- )
        notfactor
        BEGIN   '&' =token
        WHILE   _push next  notfactor _popand
        REPEAT ;

-- Recognize and translate a boolean OR / XOR
: boolOR  ( -- ) next boolterm _popor ;
: boolXOR ( -- ) next boolterm _popxor ;

-- Parse and translate a boolean expression
:NONAME ( -- )
        boolterm
        BEGIN   token orop?
        WHILE   _push
                CASE token
                  '|' OF  boolOR  ENDOF
                  '~' OF  boolXOR ENDOF
                ENDCASE
        REPEAT ; IS boolexpression
```

```
-- Parse and translate an assignment statement
: assignment ( -- )
        token> DUP 1+ ALLOCATE ?ALLOCATE DUP LOCAL name PACK DROP
        next
        S" =" matchstring
        boolexpression
        name COUNT storevariable
        name FREE ?ALLOCATE ;

-- Block statements ------------------------------------------

DEFER block

-- Recognize and translate an IF construct
: doIF ( label -- )
        0 0 LOCALS| L2 L1 L |
        next
        boolexpression
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L1 PACK DROP
        L1 COUNT DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP
        L1 COUNT _branch0
        L block
        'l' =token IF   L2 FREE ?ALLOCATE
                        next
                        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP
                        L2 COUNT _branch
                        L1 COUNT postlabel
                        L block
                ENDIF
        L2 COUNT postlabel
        S" ENDIF" matchstring

        L1 FREE ?ALLOCATE
        L2 FREE ?ALLOCATE ;

-- Recognize and translate a WHILE construct
: doWHILE ( -- )
        0 0 LOCALS| L2 L1 |
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L1 PACK DROP
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP

        next
        L1 COUNT postlabel
        boolexpression
        L2 COUNT _branch0
        L2 block
        S" ENDWHILE" matchstring
        L1 COUNT _branch
        L2 COUNT postlabel

        L1 FREE ?ALLOCATE
        L2 FREE ?ALLOCATE ;

-- Parse and translate a LOOP statement
: doLOOP ( -- )
        0 0 LOCALS| L2 L1 |
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L1 PACK DROP
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP

        next
        L1 COUNT postlabel
        L2 block
        L1 COUNT _branch
        L2 COUNT postlabel
        S" ENDLOOP" matchstring

        L1 FREE ?ALLOCATE
```

```
        L2 FREE ?ALLOCATE ;

-- Parse and translate a REPEAT statement
: doREPEAT ( -- )
        0 0 LOCALS| L2 L1 |
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L1 PACK DROP
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP

        next
        L1 COUNT postlabel
        L2 block
        S" UNTIL" matchstring
        boolexpression
        L1 COUNT _branch0
        L2 COUNT postlabel

        L1 FREE ?ALLOCATE
        L2 FREE ?ALLOCATE ;

-- Parse and translate a FOR statement
: doFOR ( -- )
        0 0 0 LOCALS| name L2 L1 |
        next
        checkident  token> checktable
        token>   DUP 1+ ALLOCATE ?ALLOCATE DUP TO name PACK DROP
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L1   PACK DROP
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2   PACK DROP

        next S" =" matchstring  expression
        _decr
        name COUNT storevariable
        S" TO" matchstring
        expression
        _push
        L1 COUNT postlabel
        name COUNT loadvariable
        _incr
        name COUNT storevariable
        L2 COUNT _pcmp+b0>
        L2 block
        L1 COUNT _branch
        L2 COUNT postlabel
        S" ENDFOR" matchstring
        _incSP

        name FREE ?ALLOCATE
        L1   FREE ?ALLOCATE
        L2   FREE ?ALLOCATE ;

-- Parse and Translate a DO Statement
: doDO ( -- )
        0 0 LOCALS| L2 L1 |
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L1 PACK DROP
        newlabel DUP 1+ ALLOCATE ?ALLOCATE DUP TO L2 PACK DROP

        next
        expression
        L1 COUNT postlabel
        _push
        L2 block
        _pop
        _decr
        L1 COUNT _branch<>0
        _decSP
        L2 COUNT postlabel
        S" ENDDO" matchstring
        _incSP
```

```
        L1 FREE ?ALLOCATE
        L2 FREE ?ALLOCATE ;

-- Recognize and translate a BREAK
: doBREAK ( label -- )
        DUP C@ 0= IF  S" No loop to break from" aborts  ENDIF
        COUNT _branch
        next ;

-- Read a single variable
: readvar ( -- ) checkident  token> checktable  token> readit  next ;

-- Process a read statement
: doread ( -- )
        next
        S" (" matchstring
        readvar
        BEGIN  ',' =token
        WHILE  next readvar
        REPEAT
        S" )" matchstring ;

-- Process a write statement
: dowrite ( -- )
        next
        S" (" matchstring
        expression writeit
        BEGIN  ',' =token
        WHILE  next expression writeit
        REPEAT
        S" )" matchstring ;

-- Recognize and translate a statement block
:NONAME ( label -- )
        LOCAL L
        scan
        BEGIN  'e' <>token
               'l' <>token AND
               'u' <>token AND
        WHILE  CASE token
                'i' OF  L doIF    ENDOF
                'w' OF  doWHILE   ENDOF
                'd' OF  doDO      ENDOF
                'L' OF  doLOOP    ENDOF
                'r' OF  doREPEAT  ENDOF
                'f' OF  doFOR     ENDOF
                'B' OF  L doBREAK ENDOF
                'R' OF  doread    ENDOF
                'W' OF  dowrite   ENDOF
                        assignment
               ENDCASE
               scan
        REPEAT ; IS block

-- Declarations ----------------------------------------------

-- Allocate storage for a variable
: alloc ( -- )
        0 LOCAL aname
        next
        token> DUP 1+ ALLOCATE ?ALLOCATE DUP TO aname PACK DROP
        'x' <>token IF  S" Variable name" expected  ENDIF
        aname COUNT checkdup
        aname COUNT 'v' addentry
        next
        '=' =token IF   S" =" matchstring
```

```
                        '-' =token IF  S" -" matchstring
                                       S" -"
                                 ELSE  S" "
                                ENDIF
                        token> $+
                        next
                 ELSE   S" 0"
                ENDIF
        aname COUNT allocatestorage
        aname FREE ?ALLOCATE ;

-- Parse and translate the global declarations
: topdecls ( -- )
        scan
        BEGIN   'v' =token
        WHILE   alloc
                BEGIN ',' =token WHILE alloc REPEAT
        REPEAT ;

-- Parse and translate a program -----------------------------

-- Initialize everything
: init ( -- )
        CLEAR Lcount
        [cnt]ST    0!
        [cnt]SType 0!
        CR getchar next ;

: TINY11 ( -- )
        init
        S" PROGRAM" matchstring  token> prog$ PACK DROP next
        header
        topdecls
        S" BEGIN" matchstring
        prolog
        C" " block
        S" END" matchstring
        epilog ;
-- -----------------------------------------------------------
```

12. Разное

ВВЕДЕНИЕ

Эта глава - второе из тех отклонений в сторону, которые не вписываются в основное направление этой обучающей серии. Я упомянул в прошлый раз, что я накопил некоторые изменения, которые должны быть внесены в структуру компилятора. Поэтому я должен был отклониться чтобы разработать новую структуру и показать ее вам.

Теперь, когда все это позади, я могу сказать что я намерен сделать прежде всего. Это не должно занять много времени и затем мы сможем вернуться в основное русло.

Некоторые люди спрашивали меня о вещах, которые предоставляют другие языки, но к которым я пока что не обращался в этой серии. Две самых больших из них - это точки с запятой и комментарии. Возможно вы тоже задумывались о них и гадали как изменился бы компилятор если бы мы работали с ними. Только для того, чтобы вы могли проследовать дальше без тревожащего чувства, что что-то пропущено, мы рассмотрим здесь эти вопросы.

ТОЧКИ С ЗАПЯТОЙ

Со времен появления Алгола точки с запятой были частью почти каждого современного языка. Все мы использовали их считая это само собой разумеющимся. Однако я полагаю, что больше ошибок компиляции происходило из-за неправильно размещенной или отсутствующей точки с запятой, чем в любом другом случае. И если бы мы получали один пенни за каждое дополнительное нажатие клавиши, использованное программистами для

набора этих маленьких мошенников, мы смогли бы оплатить национальный долг.

Будучи воспитанным на Фортране, я потратил много времени чтобы привыкнуть к использованию точек с запятой и сказать по правде я никогда полностью не понимал почему они необходимы. Так как я программирую на Паскале и так как использование точек с запятой в Паскале особенно мудрено, этот один небольшой символ является моим наибольшим источником ошибок.

Когда я начал разработку KISS, я решил критически относиться к каждой конструкции в других языках и попытаться избежать наиболее общих проблем, происходящих с ними. Это ставит точку с запятой очень высоко в моем хит-листе.

Чтобы понять роль точки с запятой, вы должны изучить краткую предисторию.

Ранние языки программирования были строчно-ориентированными. В Фортране, например, различные части утверждения занимали определенные столбцы и поля, в которых они должны были располагаться. Так как некоторые утверждения были слишком длинными для одной строки, существовал механизм "карта продолжения" ("continuation card"), позволяющий компилятору знать, что данная карта все еще является частью предыдущей строки. Механизм дожил до наших дней, хотя перфокарты теперь - дела отдаленного прошлого.

Когда появились другие языки, они также приняли различные механизмы для работы с многострочными операторами. Хороший пример - BASIC. Важно осознать, тем не менее, что механизм Фортрана был затребован не столько строчной ориентацией языка, сколько ориентацией по столбцам. В тех версиях Фортрана, где разрешен ввод в свободной форме, он больше не нужен.

Когда отцы Алгола представили этот язык, они хотели уйти от строчно-ориентированных программ подобных Фортрану или Бейсику и разрешить ввод в свободной форме. Это дало возможность связывать множество утверждений в одну строку как например:

```
a=b; c=d; e=e+1;
```

В случаях подобных этому, точка с запятой почти обязательна. Та же самая строка без точек с запятой выглядит просто "странной":

```
a=b c= d e=e+1
```

Я полагаю, что это главная... возможно единственная... причина точек с запятой: не давать программам выглядеть странно.

Но идея со связыванием множества утверждений вместе в одну строку в лучшем случае сомнительная. Это не очень хороший стиль программирования и возвращение назад к дням, когда считалось важным сэкономить карты. В наши дни CRT и выровненного кода ясность программ гораздо лучше обеспечивается сохранением раздельных утверждений. Все-таки хорошо иметь возможность множественных утверждений, но это позор - оставлять программистов в рабстве у точек с запятой только чтобы этот редкий случай не "выглядел странно".

Когда я начинал KISS, я попытался держать глаза открытыми. Я решил, что буду использовать точки с запятой когда это станет необходимо для синтаксического анализатора, но только тогда. Я рассчитывал, что это случится примерно в то время, когда я добавил возможность разложения утверждений на несколько строк. Но как вы можете видеть это никогда не случилось. Компилятор TINY совершенно счастлив анализировать наиболее сложные утверждения, разложенные на любое число строк, без точек с запятой.

Однако, есть люди, которые использовали точки с запятой так долго, что они чувствуют себя без них голыми. Я один из них. Как только KISS стал достаточно хорошо определен, я попробовал написать на этом языке несколько программ-примеров. Я обнаружил, к своему ужасу, что пытаюсь в любом случае расставлять точки с запятой. Так что теперь я стою перед перспективой нового потока ошибок компилятора, вызванных нежелательными

точками с запятой. Тьфу!

Возможно более важно, что есть читатели, которые разрабатывают свои собственные языки, которые могут включать точки с запятой или которые хотят использовать методы из этого руководства для компиляции стандартных языков типа C. В обоих случаях, нам необходима возможность работать с точками с запятой.

СИНТАКСИЧЕСКИЙ САХАР

Вся эта дисскуссия поднимает вопрос "синтаксического сахара"... конструкций, которые добавлены к языку не потому, что они нужны, но потому, что они заставляют программы выглядеть правильно для программиста. В конце концов, хорошо иметь маленький простой компилятор, но от него было бы мало пользы если бы полученный язык был закодированным или сложным для программирования. На ум приходит язык FORTH. Если мы можем добавить в язык возможности, которые делают программы более легкими для чтения и понимания, и если эти возможности предохраняют программиста от ошибок, тогда мы бы сделали это. Особенно если эти конструкции не добавляют слишком много сложности в язык или его компилятор.

Как пример можно было бы рассмотреть точку с запятой, но существуют множество других, такие как "THEN" в операторе IF, "DO" в операторе WHILE или даже утверждение "PROGRAM" которое я убрал из TINY. Ни один из этих токенов не добавляет много к синтаксису языка... компилятор может выяснить, что происходит и без них. Но некоторые люди чувствуют, что они улучшают читаемость программ и это может быть очень важно.

Существуют две школы мысли на эту тему, которые хорошо представлены двумя из наших наиболее популярных языков C и Pascal.

По мнению минималистов весь подобный сахар должен быть выброшен. Они доказывают, что это загромождает язык и увеличивает число нажатий клавиш, которое должен сделать программист. Возможно более важно, что каждый дополнительный токен или ключевое слово представляет собой ловушку, лежащую в ожидании невнимательного программиста. Если вы не учтете токен, пропустите его или сделаете в нем орфографическую ошибку, компилятор достанет вас. Поэтому эти люди утверждают, что лучший подход - избегать таких вещей. Этот народ предпочитает язык типа C, который имеет минимум ненужных ключевых слов и пунктуаций.

Те, кто относятся к другой школе, предпочитают язык Pascal. Они доказывают, что необходимость набрать несколько дополнительных символов - это малая цена за удобочитаемость. В конце концов, люди тоже должны читать программы. Их самый лучший аргумент в том, что каждая такая конструкция является возможностью сообщить компилятору, что вы действительно хотите того, что сказали. Сахарные токены служат в качестве полезных ориентиров, помогающих вам не сбиться с пути..

Различия хорошо представлены этими двум языками. Наиболее часто слышимая претензия к C - что он слишком прощающий. Когда вы делаете ошибку в C, ошибочный код часто является другой допустимой конструкцией C. Так что компилятор просто счастливо продолжает компиляцию и оставляет вам нахождение ошибок в течение отладки. Я предполагаю именно поэтому отладчики так популярны среди C программистов.

С другой стороны, если компилируется программа на Паскале, вы можете быть вполне уверены, что программа будет делать то, что вы ей сказали. Если имеется ошибка во время выполнения, возможно это ошибка разработки.

Самым лучшим примером полезного сахара является непосредственно точка с запятой. Рассмотрите фрагмент кода:

```
a=1+(2*b+c)    b...
```

Так как нет никакого оператора, соединяющего токен 'b' с остальной частью выражения, компилятор заключит, что выражение заканчивается на ')' а 'b' - это начало нового

утверждения. Но предположим, что я просто пропустил предполагаемый оператор и в действительности хотел сказать:

```
a=1+(2*b+c)*b...
```

В этом случае компилятор выдаст ошибку, хорошо, но она не будет очень осмысленной, так как он будет ожидать знак '=' после 'b', который в действительности не должен быть там.

Если, наооборот, я вставлю точку с запятой после 'b', тогда не может остаться сомнений где, как я предполагаю, заканчивается утверждение. Синтаксический сахар, т.о., может служить очень полезной цели, предоставляя некоторую дополнительную подстраховку что мы остаемся на правильном пути.

Я нахожусь где-то посередине между этими подходами. Я склоняюсь к преимуществам Паскалевской точки зрения... я был бы очень доволен находить свои ошибки во время компиляции а не во время выполнения. Но я также ненавижу просто бросаться словами без явной причины как в COBOL. Пока что я последовательно выкинул большинство Паскалевского сахара из KISS/TINY. Но я конечно не испытываю сильных чувств к любому способу и я также могу видеть значение разбрасывания небольшого количества сахара только для дополнительной подстраховки, которую он дает. Если вам нравится этот последний подход, такие вещи легко добавить. Только запомните, что как и точка с запятой каждая ложка сахара - это что-то, что может потенциально привести к ошибке компиляции при ее пропуске.

РАБОТА С ТОЧКАМИ С ЗАПЯТОЙ

Есть два различных способа работы с точками с запятой используемые в популярных языках. В Паскале точка с запятой расценивается как разделитель операторов. Точка с запятой не требуется после последнего утверждения в блоке. Синтаксис:

```
<block> ::= <statement> ( ';' <statement>)*

<statement> ::= <assignment> | <if> | <while> ... | null
```

(пустое утверждение важно!)

Паскаль также определяет некоторые точки с запятой в других местах, таких как после утверждения PROGRAM.

В C и Ada, с другой стороны, точка с запятой рассматриватся как терминатор операторов и следует после всех утверждений (с некоторыми смущающими и путающими исключениями). Синтаксис для них простой:

```
<block> ::= ( <statement> ';')*
```

Из двух синтаксисов, синтаксис Паскаля внешне выглядит более рациональным, но опыт показал, что он ведет к некоторым странным трудностям. Люди так привыкают ставить точку с запятой после каждого утверждения, что они также предпочитают ставить ее и после последнего утверждения в блоке. Это обычно не приносит какого-либо вреда... она просто обрабатывается как пустое утверждение. Многие программисты на Паскале, включая вашего покорного слугу, делают точно также. Но есть одно место, в котором вы абсолютно не можете поставить точку с запятой - прямо перед ELSE. Это маленький подводный камень стоил мне множества дополнительных компиляций, особенно когда ELSE добавляется к существующему коду. Так что выбор C/Ada оказывается лучше. Очевидно, Никлаус Вирт думает также: в его Modula-2 он отказался от Паскалевского подхода.

Имея эти два синтаксиса, легко (теперь, когда мы реорганизовали синтаксический анализатор!) добавить эти возможности в наш анализатор. Давайте сначала возьмем последний случай, так как он проще.

Для начала я упростил программу представив новую подпрограмму распознавания:

```
-- -------------------------------------------------------------
-- Match a semicolon
: semi ( -- ) S" ;" matchstring ;
-- -------------------------------------------------------------
```

Эта процедура очень похожа на наш старый Match. Она требует чтобы следующим токеном была точка с запятой. Найдя его, она переходит к следующему.

Так как точка с запятой следует за утверждением, процедура Block почти единственная, которую мы должны изменить:

```
-- -------------------------------------------------------------
-- Parse and translate a block of statements
:NONAME ( -- )
        scan
        BEGIN  token 'e' <>
               token 'l' <> AND
        WHILE  CASE token
                'i' OF  doIF       ENDOF
                'w' OF  doWHILE    ENDOF
                'R' OF  doread     ENDOF
                'W' OF  dowrite    ENDOF
                'x' OF  assignment ENDOF
               ENDCASE
               semi
               scan
        REPEAT ; IS block
-- -------------------------------------------------------------
```

Внимательно взгляните на тонкие изменения в операторе case. Вызов Assigment теперь ограничивается проверкой Token. Это позволит избежать вызова Assigment когда токен является точкой с запятой (что случается когда утверждение пустое).

Так как объявления - тоже утверждения, мы также должны добавить вызов Semi в процедуру TopDecl:

```
-- -------------------------------------------------------------
-- Parse and translate global declarations
: topdecls ( -- )
        scan
        BEGIN   token 'v' =
        WHILE   alloc
                BEGIN token ',' = WHILE alloc REPEAT
                semi
        REPEAT ;
-- -------------------------------------------------------------
```

Наконец нам нужен вызов для утверждения PROGRAM:

```
-- -------------------------------------------------------------
-- Main Program
: TINY12 ( -- )
        init
        S" PROGRAM" matchstring
        token$ COUNT prog$ PACK DROP
        semi
        header
        topdecls
        S" BEGIN" matchstring
        prolog
        C" " block
        S" END" matchstring
        epilog ;
-- -------------------------------------------------------------
```

Проще некуда. Испробуйте это с копией TINY и скажите как вам это нравится.

Версия Паскаля немного сложнее, но она все равно требует только небольших изменений и то только в процедуре Block. Для максимальной простоты давайте разобьем процедуру на две части. Следующая процедура обрабатывает только одно утверждение:

```
-- ---------------------------------------------------------------
-- Parse and translate a single statement
: statement ( -- )
        scan
        CASE token
          'i' OF doIF       ENDOF
          'w' OF doWHILE    ENDOF
          'R' OF doRead     ENDOF
          'W' OF doWrite    ENDOF
          'x' OF Assignment ENDOF
        ENDCASE ;
-- ---------------------------------------------------------------
```

Используя эту процедуру мы можем переписать Block так:

```
-- ---------------------------------------------------------------
-- Parse and translate a block of statements
:NONAME ( -- )
        statement
        BEGIN  token ';' =
        WHILE  next statement
        REPEAT ; IS block
-- ---------------------------------------------------------------
```

Это, уверен, не повредило, не так ли? Теперь мы можем анализировать точки с запятой в Паскаль-подобном стиле.

КОМПРОМИСС

Теперь, когда мы знаем как работать с точками с запятой, означает ли это, что я собираюсь поместить их в KISS/TINY? И да и нет. Мне нравится дополнительный сахар и защита, которые приходят с уверенным знанием, где заканчиваются утверждения. Но я не изменил своей антипатии к ошибкам компиляции, связанным с точками с запятой.

Так что я придумал хороший компромис: сделаем их необязательными!

Рассмотрите следующую версию Semi:

```
-- ---------------------------------------------------------------
-- Match a semicolon
: semi ( -- ) token ';' = IF next ENDIF ;
-- ---------------------------------------------------------------
```

Эта процедура будет принимать точку с запятой всякий раз, когда вызвана, но не будет настаивать на ней. Это означает, что когда вы решите использовать точки с запятой, компилятор будет использовать дополнительную информацию чтобы удержаться на правильном пути. Но если вы пропустите одну (или пропустите их всех) компилятор не будет жаловаться. Лучший из обоих миров.

Поместите эту процедуру на место в первую версию вашей программы (с синтаксисом для C/Ada) и вы получите TINY Version 1.2.

КОММЕНТАРИИ

Вплоть до этого времени я тщательно избегал темы комментариев. Вы могли бы подумать, что это будет простая тема... в конце концов компилятор совсем не должен иметь дела с комментариями; он просто должен игнорировать их. Чтож, иногда это так.

Насколько простыми или насколько трудными могут быть комментарии, зависит от выбранного вами способа их реализации. В одном случае, мы можем сделать так, чтобы эти комментарии перехватывались как только они поступят в компилятор. В другом, мы можем обрабатывать их как лексические элементы. Станет интереснее когда вы рассмотрите вещи, типа разделителей комментариев, содержащихся в строках в кавычках.

ОДНОСИМВОЛЬНЫЕ РАЗДЕЛИТЕЛИ

Вот пример. Предположим, мы принимаем стандарт Turbo Pascal и используем для комментариев фигурные скобки. В этом случае мы используем односимвольные разделители, так что наш анализ немного проще.

Один подход состоит в том, чтобы удалять комментарии как только мы встретим их во входном потоке, т.е. прямо в процедуре GetChar. Чтобы сделать это сначала измените имя GetChar на какое-нибудь другое, скажем GetCharX. (На всякий случай запомните, это будет временное изменение, так что лучше не делать этого с вашей единственной копией TINY. Я полагаю вы понимаете, что вы всегда должны делать эти эксперименты с рабочей копией).

Теперь нам нужна процедура для пропуска комментариев. Так что наберите следующее:

```
-- ---------------------------------------------------------------
-- Skip a comment field
: skipcomment ( -- )
        BEGIN  Look '}' <>
        WHILE  getcharx
        REPEAT getcharx ;
-- ---------------------------------------------------------------
```

Ясно, что эта процедура будет просто считывать и отбрасывать символы из входного потока, пока не найдет правую фигурную скобку. Затем она считывает еще один символ и возвращает его в Look.

Теперь мы можем написать новую версию GetChar, которая вызывает SkipComment для удаления комментариев:

```
-- ---------------------------------------------------------------
-- Get character from input stream
-- Skip any comments
: getchar ( -- )
        getcharx
        Look '{' = IF  skipcomment  ENDIF ;
-- ---------------------------------------------------------------
```

Наберите этот код и испытайте его. Вы обнаружите, что вы действительно можете вставлять комментарии везде, где захотите. Комментарии никогда даже не попадут в синтаксический анализатор... каждый вызов GetChar просто возвращает любой символ, не являющийся частью комментария.

Фактически, хотя этот метод делает свое дело и может даже совершенно удовлетворять вас, он делает свою работу немного слишком хорошо. Прежде всего, большинство языков программирования определяет, что комментарии должны быть обработаны как пробелы, так как комментарии не могут быть вложены, скажем, в имя переменной. Эта текущая версия не заботится о том, где вы помещаете комментарии.

Во-вторых, так как остальная часть синтаксического анализатора не может даже получить символ '{', вам не будет позволено поместить его в строку в кавычках.

Однако, прежде чем вы отвернете свой нос от такого упрощенного решения, я должен подчеркнуть, что столь уважаемый компилятор, как Turbo Pascal, также не позволит использовать '{' в строке в кавычках. Попробуйте. Относительно комментариев, вложенных в идентификатор, я не могу представить чтобы кто-то захотел сделать подобные вещи, так что

вопрос спорен. Для 99% всех приложений то что я показал, будет работать просто отлично.

Но, если вы хотите быть щепетильным в этом вопросе и придерживаться стандартного обращения, тогда нам нужно переместить место перехвата немного ниже.

Чтобы сделать это сперва верните GetChar на старое место и измените имя, вызываемое в SkipComment Затем, давайте добавим левую фигурную скобку как возможный символ незаполненного пространства:

```
-- --------------------------------------------------------------
-- Recognize white space
: white? ( char -- tf )
        DUP  BL  =
        OVER Tab = OR
        OVER ^M  = OR
        OVER ^J  = OR
        SWAP '{' = OR ;
-- --------------------------------------------------------------
```

Теперь мы можем работать с комментариями в процедуре SkipWhite:

```
-- --------------------------------------------------------------
-- Skip over leading white space
: skipwhite ( -- )
        BEGIN  Look white?
        WHILE  Look '{' = IF skipcomment ELSE getchar ENDIF
        REPEAT ;
-- --------------------------------------------------------------
```

Обратите внимание, что SkipWhite написан так, что мы пропустим любую комбинацию незаполненного пространства и комментариев в одном вызове.

Протестируйте компилятор. Вы обнаружите, что он позволит комментариям служить разделителями токенов. Заслуживает внимания, что этот подход также дает нам возможность обрабатывать фигурные скобки в строках в кавычках, так как внутри этих строк мы не будем проверять или пропускать пробелы.

Остался последний вопрос: вложенные комментарии. Некоторым программистам нравится идея вложенных комментариев так как это позволяет комментировать код во время отладки. Код, который я дал здесь не позволит этого и, снова, не позволит и Turbo Pascal.

Но исправить это невероятно просто. Все, что нам нужно - сделать SkipComment рекурсивной:

```
-- --------------------------------------------------------------
-- Skip a comment field
: skipcomment ( -- )
        BEGIN   Look '}' <>
        WHILE   getchar
                Look '{' = IF  RECURSE  ENDIF
        REPEAT  getchar ;
-- --------------------------------------------------------------
```

Готово. Настолько утонченный обработчик комментариев, какой вам когда-либо может понадобиться.

МНОГОСИМВОЛЬНЫЕ РАЗДЕЛИТЕЛИ

Все это хорошо для случаев, когда комментарии ограничены одиночными символами, но как быть с такими случаями как C или стандартный Pascal, где требуются два символа? Хорошо, принцип все еще тот же самый, но мы должны совсем немного изменить наш подход. Я уверен, что вы не удивитесь узнав, что это более сложный случай.

Для многосимвольной ситуации проще всего перехватывать левый ограничитель в

GetChar. Мы можем "токенизировать" его прямо здесь, заменяя его одиночным символом.

Давайте условимся, что мы используем ограничители C '/*' и '*/'. Сначала мы должны возвратиться к методу 'GetCharX'. В еще одной копии вашего компилятора переименуйте GetChar в GetCharX и затем введите следующую новую процедуру GetChar:

```
-- ----------------------------------------------------------------
-- Read new character.  intercept '/*'
0 VALUE TempChar
: getchar ( -- )
   TempChar BL <>
      IF  TempChar TO Look
          BL TO TempChar
    ELSE  getcharx
          Look '/' = IF  EKEY TO TempChar
                         TempChar '*'
                          = IF  '{' TO Look
                                BL  TO TempChar
                         ENDIF
                  ENDIF
   ENDIF ;
-- ----------------------------------------------------------------
```

Как вы можете видеть эта процедура перехватывает каждое появление '/'. Затем она исследует следующий символ в потоке. Если это символ '*', то мы нашли начало комментария и GetChar возвратит его односимвольный заменитель. (Для простоты я использую тот же самый символ '{' как я делал для Паскаля. Если бы вы писали компилятор C, вы без сомнения захотели бы использовать какой-то другой символ, не используемый где-то еще в C. Выберите что вам нравится... даже $FF, что-нибудь уникальное).

Если символ, следующий за '/' не '*', тогда GetChar прячет его в новой глобальной переменной TempChar и возвращает '/'.

Обратите внимание, что вы должны объявить эту новую переменную и присвоить ей значение ' '. Мне нравится делать подобные вещи с использование конструкции "типизированная константа" в Turbo Pascal:

```
BL VALUE TempChar
```

Теперь нам нужна новая версия SkipComment:

```
-- ----------------------------------------------------------------
-- Skip a comment field
: skipcomment ( -- )
        BEGIN
          BEGIN
            getcharx
            Look '*' =
          UNTIL
          getcharx
          Look '/' =
        UNTIL
        getchar ;
-- ----------------------------------------------------------------
```

Обратите внимание на несколько вещей: прежде всего нет необходимости изменять функцию IsWhite и процедуру SkipWhite так как GetChar возвращает токен '{'. Если вы измените этот символ токена, тогда конечно вы также должны будете изменить символ в этих двух подпрограммах.

Во-вторых, заметьте, что SkipComment вызывает в своем цикле не GetChar а GetCharX. Это означает, что завершающий '/' не перехватывается и обрабатывается SkipComment. В-третьих, хотя работу выполняет процедура GetChar, мы все же можем работать с символами комментариев вложенными в строки в кавычках, вызывая GetCharX вместо GetChar пока мы

находимся внутри строки. Наконец, заметьте, что мы можем снова обеспечить вложенные комментарии добавив одиночное утверждение в SkipComment, точно также как мы делали прежде.

ОДНОСТОРОННИЕ КОММЕНТАРИИ

Пока что я показал вам как работать с любыми видами комментариев, ограниченных слева и справа. Остались только односторонние комментарии подобные используемым в ассемблере или Ada, которые завершаются концом строки. На практике этот способ проще. Единственная процедура, которая должна быть изменена - SkipComment, которая должна теперь завершаться на символе переноса строки:

```
-- -------------------------------------------------------------
-- Skip a comment field
: skipcomment ( -- )
        BEGIN
          getcharx;
          Look ^M =
        UNTIL
        getchar ;
-- -------------------------------------------------------------
```

Если ведущий символ - одиночный, как ";" в ассемблере, тогда мы по существу все сделали. Если это двухсимвольный токен, как "--" из Ada, нам необходимо только изменить проверки в GetChar. В любом случае это более легкая проблема чем двухсторонние комментарии.

ЗАКЛЮЧЕНИЕ

К этому моменту у нас есть возможность работать и с комментариями и точками с запятой, так же как и с другими видами синтаксического сахара. Я показал вам несколько способов работы с каждым из них, в зависимости от желаемых соглашений. Остался единственный вопрос - какие из этих соглашений мы должны использовать в KISS/TINY?(tiny12.frt)

По причинам, которые я высказал по ходу дела, я выбираю следующее:

- Точки с запятой - терминаторы а не разделители.
- Точки с запятой необязательны.
- Комментарии ограничены фигурными скобками.
- Комментарии могут быть вложенными.

Поместите код, соответствующий этим случаям в вашу копию TINY. Теперь у вас есть TINY Version 1.2.

Теперь, когда мы разрешили эти побочные проблемы, мы можем наконец возвратиться в основное русло. В следующей главе мы поговорим о процедурах и передаче параметров и добавим эти важные возможности в TINY. Увидимся.

13. Процедуры

ВВЕДЕНИЕ

Наконец-то мы принимаемся за хорошую главу!

К этому моменту мы изучили почти все основные особенности компиляторов и синтаксического анализа. Мы узнали как транслировать арифметические выражения, булевы выражения, управляющие конструкции, объявления данных и операторы ввода/вывода. Мы определили язык TINY 1.3, который воплощает все эти возможности, и написали

элементарный компилятор, который может их транслировать. Добавив файловый ввод/вывод мы могли бы действительно иметь работающий компилятор, способный производить выполнимые объектные файлы из программ, написанных на TINY. С таким компилятором мы могли бы писать простые программы, способные считывать целочисленные данные, выполнять над ними вычисления и выводить результаты.

Все это хорошо, но то, что у нас есть, это все еще только игрушечный язык. Мы не можем считывать и выводить даже одиночные символы текста и у нас все еще нет процедур.

Эти возможности, которые будут обсуждены в следующих двух главах, так сказать отделят мужчин от игрушек. "Настоящие" языки имеют более одного типа данных и они поддерживают вызовы процедур. Более чем любые другие, именно эти две возможности дают языку большую часть его характера и индивидуальности. Как только мы предоставим их, наши языки, TINY и его преемники, перестанут быть игрушками и получат характер настоящих языков, пригодных для серъезного программирования.

В течение нескольких предыдущих глав я обещал вам урок по этим двум важным темам. Каждый раз появлялись другие проблемы, которые требовали отклонения и работы с ними. Наконец у нас появилась возможность оставить все эти проблемы в покое и вернуться в основное русло. В этой главе я охвачу процедуры. В следующий раз мы поговорим об основных типах данных.

ПОСЛЕДНЕЕ ОТКЛОНЕНИЕ

Эта глава была необычайно трудной для меня. Причина не имеет никакого отношения непосредственно к теме... я знал, о чем хотел рассказать уже какое-то время, и фактически я представил большинство из этого на Software Development '89, в феврале. Больше это имело отношение к подходу. Позвольте мне объяснить.

Когда я впервые начал эту серию, я сказал вам, что мы будем использовать некоторые "приемы" чтобы упростить себе жизнь и позволить нам получить общее представление не вдаваясь слишком подробно в детали. Среди этих приемов была идея рассмотрения отдельных частей компилятора в отдельные моменты времени, т.е. выполнения экспериментов, использующих Cradle как основу. Когда, например, мы исследовали выражения мы работали только с этой частью теории компиляции. Когда мы исследовали управляющие структуры, мы писали различные программы, все еще основанные на Cradle, для выполнения этой части. Мы включили эти понятия в полный язык довольно недавно. Эти методы служили нам действительно очень хорошо и привели нас к разработке компилятора для TINY версии 1.3.

Вначале, когда я начал этот урок, я попытался основываться на том, что мы уже сделали и просто добавлять новые возможности в существующий компилятор. Это оказалось немного неудобным и сложным... слишком, чтобы удовлетворить меня.

В конце концов я выяснил почему. В этой серии экспериментов я отказался от очень полезного метода, который позволил нам добраться до сюда, и без особой на то нужды я переключился на новый метод работы, который включал в себя пошаговые изменения в полной версии компилятора TINY.

Вы долны понять, что то, чем мы здесь занимаемся немного уникально. Существует ряд статей таких как статьи по Small C от Кейна и Хендрикса, которые представляли законченный компилятор для одного языка или другого. Это другое. В этой обучающей серии вы наблюдаете за моей разработкой и реализацией и языка и компилятора в реальном режиме времени.

В экспериментах, которые я проводил при подготовке к этой статье, я пробовал вносить изменения в компилятор TINY таким способом, что на каждом шаге мы бы все еще имели настоящий, работающий компилятор. Другими словами, я сделал попытку инкрементального расширения языка и его компилятора в то же самое время объясняя вам, что я делал.

Это оказалось тяжелым делом! В конце-концов я понял, что глупо было и пытаться.

Достигнув столького используя идею маленьких экспериментов, основанных на односимвольных токенах и простых, специализированных программах, я отказался от них в пользу работы с полным компилятором. Это не сработало.

Поэтому мы собираемся возвратиться к своим корням, так сказать. В этой и следующей главах я буду снова использовать односимвольные токены для исследования концепции процедур, освобожденный от другого багажа, накопленного нами на предыдущих уроках. Фактически, я даже не буду пытаться в конце этого урока обьединить конструкции в компилятор TINY. Мы оставим это на потом.

В конце концов на этот раз вам не понадобится что-то большее, так что давайте не будем больше тратить времени зря и двинемся вперед.

ОСНОВЫ

Все современные центральные процессоры предоставляют прямую поддержку вызовов процедур и 68000 не исключение. Для 68000 вызов - BSR (PC-относительная версия) или JSR, и возвращение RTS. Все что мы должны сделать это организовать для компилятора выдачу этих команд в соответствующих местах.

В действительности есть три вещи, которые мы должны рассмотреть. Одна из них - механизм вызова/возврата. Вторая - механизм определения процедур. И, наконец, вопрос передачи параметров в вызываемую процедуру. Ни одна из этих вещей не является в действительности очень сложной и мы можем конечно позаимствовать то, что сделано в других языках... нет необходимости заново изобретать колесо. Из этих трех вопросов передача параметров займет большую часть нашего внимания просто потому что здесь существует много возможностей.

ОСНОВА ДЛЯ ЭКСПЕРИМЕНТОВ

Как всегда нам понадобится некоторое программное обеспечение, которое послужит нам как основание для того, что мы делаем. Нам не нужна полная версия компилятора TINY но нам нужна достаточная его часть для того, чтобы некоторые конструкции были представлены. В частности, нам нужна по крайней мере возможность обрабатывать утверждения некоторых видов и объявления данных.

Программа, показанная ниже является такой основой. Это остаточная форма TINY с односимвольными токенами. Она имеет объявления данных, но только в их самой простейшей форме... никаких списков или инициализаторов. Имеются операции присваивания, но только вида

```
<ident> = <ident>
```

Другими словами, единственным допустимым выражением является одиночное имя переменной. Нет никаких управляющих конструкций... единственным допустимым утверждением является присваивание.

Большую часть программы составляют просто подпрограммы из стандартного Cradle. Я показал ее здесь полностью только для того, чтобы быть уверенным что все мы начинаем с одного места:(calls1.frt

```
-- --------------------------------------------------------------
-- Variable declarations ------------------------------------

0 VALUE Look                            -- lookahead character

CREATE ST #26 CHARS ALLOT               -- symbol table

-- Get type of symbol
: typeof ( char -- ) 'A' - ST + C@ ;

-- Look for symbol in table
```

```
: intable ( char -- ) typeof 0<> ;

-- Tools ----------------------------------------------------

: getchar  ( -- )          EKEY TO Look ;                      -- read new
character from input stream
: error    ( c-addr u -- ) CR ^G EMIT ." Error: " TYPE ." ." ;  -- report an error
: aborts   ( c-addr u -- ) error ABORT ;                        -- report error and
halt
: expected ( c-addr u -- ) S"  expected" $+ aborts ;            -- report what was
expected
: alpha?   ( char -- tf )  >UPC 'A' 'Z' 1+ WITHIN ;             -- recognize an
alpha character
: digit?   ( char -- tf )  '0' '9' 1+ WITHIN ;                  -- recognize a
decimal digit
: alnum?   ( char -- tf )  DUP alpha? SWAP digit? OR ;          -- recognize
alphanumeric
: white?   ( char -- tf )  DUP  BL =  SWAP Tab =  OR ;          -- Recognize white
space
: emits    ( c-addr u -- ) Tab EMIT TYPE ;                      -- output a string
with tab
: emitdln  ( c-addr u -- ) CR TYPE ;                            -- output a string
with crlf
: emitln   ( c-addr u -- ) CR emits ;                           -- output a string
with tab and crlf

-- Skip over EOL
: fin ( -- ) Look ^M = IF  getchar  Look ^J = IF  getchar  ENDIF ENDIF ;

-- Skip a comment field
: skipcomment RECURSIVE ( -- )
        BEGIN   Look '}' <>
        WHILE   getchar
                Look '{' = IF  skipcomment  ENDIF
        REPEAT  getchar ;

-- Skip white space
: skipwhite ( -- )
        BEGIN  Look white?
        WHILE  Look '{' = IF  skipcomment  ELSE  getchar  ENDIF
        REPEAT ;

-- Abort, reporting an offending item
: name.aborts ( c-addr1 u1 char )
        S" " ROT CHAR-APPEND &' CHAR-APPEND S"  `" 2SWAP $+ $+ aborts ;

: .undefined ( char -- ) >S S" Undefined identifier" S> name.aborts ;
: .duplicate ( char -- ) >S S" Duplicate identifier" S> name.aborts ;

-- Is identifier in the symbol table?
: checktable ( char -- )
        DUP intable    IF DROP EXIT ENDIF .undefined ;

-- Is identifier already in symbol table?
: checkdup ( char -- )
        DUP intable 0= IF DROP EXIT ENDIF .duplicate ;

-- Add a new entry to symbol table
: addentry ( char type -- )
        OVER checkdup
        SWAP 'A' - ST + C! ;

: checkvar ( char -- )
        DUP checktable
        typeof 'v' <> IF  >S S" ' is not a variable" S" `"
                          S> CHAR-APPEND 2SWAP $+ aborts
                     ENDIF ;
```

```
-- Match a specific input character
: match ( char -- )
        DUP Look = IF  DROP getchar skipwhite
                 ELSE  S" `" ROT CHAR-APPEND &' CHAR-APPEND expected
                ENDIF ;

-- Get an identifier
: getname ( -- char )
        Look alpha? 0= IF  S" Identifier" expected  ENDIF
        Look >UPC  getchar skipwhite ;

-- Get a number
: getnum ( -- num )
        Look digit? 0= IF  S" Number" expected  ENDIF
        Look  getchar skipwhite ;

-- Code generation ----------------------------------------------

-- Load primary register from variable
: loadvariable  ( char -- )
        DUP checkvar
        S" " ROT CHAR-APPEND S"  dword-ptr -> eax  mov," $+ emitln ;

-- Store primary register in variable
: storevariable ( char -- )
        DUP checkvar
        S" eax -> " ROT CHAR-APPEND S"  dword-ptr  mov," $+ emitln ;

-- Expressions --------------------------------------------------

-- Parse and translate an expression
: expression ( -- )  getname loadvariable ;

-- Parse and translate an assignment statement
: assignment ( -- )  getname >R  '=' match  expression  R> storevariable ;

-- Block statements ---------------------------------------------

: doblock    ( -- )  BEGIN  Look 'e' <>  WHILE  assignment fin  REPEAT ;
: beginblock ( -- )  'b' match fin doblock 'e' match fin ;

-- Declarations -------------------------------------------------

-- Allocate storage for a variable
: alloc ( char -- )
        >R R@ 'v' addentry
        S" CREATE " R> CHAR-APPEND S"  0 ," $+ emitdln ;

: decl ( -- )  'v' match  getname alloc ;

-- Parse and translate the global declarations
: topdecls ( -- )
        BEGIN   Look 'b' <>
        WHILE   CASE Look
                  'v' OF  decl  ENDOF
                          S" Unrecognized keyword" Look name.aborts
                ENDCASE
                fin
        REPEAT ;

-- Parse and translate a program -------------------------------

-- Initialize everything
: init  ( -- ) ST #26 ERASE  CR getchar skipwhite ;
: calls ( -- ) init topdecls beginblock ;
-- -------------------------------------------------------------
```

Обратите внимание, что у нас есть таблица идентификаторов и есть логика для проверки допустимости имени переменной. Также стоит обратить внимание на то, что я включил код, который вы видели ранее для поддержки пробелов и переносов строк. Наконец заметьте, что основная программа ограничена как обычно операторными скобками BEGIN-END.

Если вы скопировали программу в Turbo, первым делом нужно откомпилировать ее и удостовериться что она работает. Сделайте несколько объявлений а затем блок begin. Попробуйте что-нибудь вроде:

```
va              (for VAR A)
vb              (for VAR B)
vc              (for VAR C)
b               (for BEGIN)
a=b
b=c
e.              (for END.)
```

Как обычно, вы должны сделать некоторые преднамеренные ошибки и проверить, что программа правильно их отлавливает.

ОБЪЯВЛЕНИЕ ПРОЦЕДУРЫ

Если вы удовлетворены, как работает наша маленькая программа, тогда пришло время поработать с процедурами. Так как мы еще не говорили о параметрах мы начнем с рассмотрения только таких процедур которые не имеют списка параметров.

Для начала, давайте рассмотрим простую программу с процедурой и подумаем о коде который мы хотели бы увидеть для нее сгенерированным:

```
PROGRAM FOO;
.
.
PROCEDURE BAR;                     BAR:
BEGIN                                   .
.                                       .
.                                       .
END;                                    ret,

BEGIN { MAIN PROGRAM }             MAIN:
.                                       .
.                                       .
BAR;                                    BAR offset NEAR call,
.                                       .
.                                       .
END.                                    END MAIN
```

Здесь я показал конструкции высокоуровневого языка слева и желаемый ассемблерный код справа. Прежде всего заметьте, что здесь несомненно нам не нужно генерировать много кода! Для большей части процедуры и основной программы наши существующие конструкции позаботятся о генерируемом коде.

Ключ к работе с телом процедуры - понимание того, что хотя процедура может быть очень длинной, ее объявление в действительности не отличается от объявления переменной. Это просто еще один вид объявлений. Мы можем записать БНФ:

```
<declaration> ::= <data decl> | <procedure>
```

Это означает, что можно легко изменить TopDecl для работы с процедурами. Как насчет синтаксиса процедуры? Хорошо, вот предлагаемый синтаксис, который по существу такой же как и в Pascal:

```
<procedure> ::= PROCEDURE <ident> <begin-block>
```

Здесь практически не требуется никакой генерации кода., кроме генерации внутри блока begin. Мы должны только выдать метку в начале процедуры и RTS в конце.

Вот требуемый код:

```
-- --------------------------------------------------------------
-- Parse and translate a procedure declaration
: doProc ( -- )
	'p' match
	getname >R fin
	R@ 'p' addentry
	R> procheader
	beginblock
	_ret ;
-- --------------------------------------------------------------
```

Обратите внимание, что я добавил новую подпрограмму генерации кода Return, которая просто выдает инструкцию RTS. Создание этой подпрограммы "оставлено как упражнение студенту".

Для завершения этой версии добавьте следующую строку в оператор Case в DoBlock.

```
		'p' OF doProc ENDOF
```

Я должен упомянуть, что эта структура для объявлений и БНФ, которая управляет ей, отличается от стандартного Паскаля. В определении Паскаля от Дженсена и Вирта объявления переменных и, фактически, все виды объявлений, должны следовать в определенном порядке, т.е. метки, константы, типы, переменные, процедуры и основная программа. Чтобы следовать такой схеме, мы должны разделить два объявления и написать в основной программе что-нибудь вроде:

```
doVars
doProcs
doMain
```

(calls2.frt) Однако, большинство реализаций Паскаля, включая Turbo, не требуют такого порядка и позволяют вам свободно перемешивать различные объявления до тех пор пока вы не попробуете обратиться к чему-то прежде, чем это будет объявлено. Хотя это может быть больше эстетическим удовлетворением объявлять все глобальные переменные на вершине программы, конечно не будет никакого вреда от того, чтобы разрешить расставлять их в любом месте. Фактически, это может быть даже немного полезно, в том смысле, что это даст нам возможность выполнять небольшое элементарное скрытие информации. Переменные, к которым нужно обращатся только из основной программы, к примеру, могут быть объявлены непосредственно перед ней и будут таким образом недоступны для процедур.

Испытайте эту новую версию. Заметьте, что мы можем объявить так много процодур, как захотим (до тех пор, пока не исчерпаем односимвольные имена!) и все метки и RTS появятся в правильных местах.

Здесь стоит заметить, что я не разрешаю вложенные процедуры. В TINY все процедуры должны быть объявлены на глобальном уровне, так же как и в C. На Форуме Компьютерных Языков на CompuServe по этому поводу возникла порядочная дискуссия. Оказывается, существует значительная расплата сложностью которая должна быть заплачена за роскошь использования вложенных процедур. Более того, эта расплата платится во время выполнения, так как должен быть добавлен дополнительный код, который будет выполняться каждый раз когда процедура вызывается. Я также охотно верю что вложение это не очень хорошая идея просто на том основании, что я видел слишком много злоупотреблений этой возможностью. Прежде, чем сделать следующий шаг, также стоит обратить внимание на то, что "основная програма" в ее текущем состоянии незавершена, так как она не имеет метки и утверждения END. Давайте исправим эту небольшую оплошность:

```
-- --------------------------------------------------------------
-- Parse and translate a main program
: doMain ( -- )
        'b' match fin
        prolog  doBlock  epilog ;
-- --------------------------------------------------------------
.
.
.
-- --------------------------------------------------------------
-- Main Program
: TINY ( -- ) init topdecls doMain ;
-- --------------------------------------------------------------
```

Обратите внимание, что DoProc и DoMain не совсем симметричны. DoProc использует вызов BeginBlock тогда как DoMain нет. Это из-за того, что начало процедуры определяется по ключевому слову PROCEDURE (в данном случае сокращенно 'p'), тогда как основная программа не имеет никакого другого ключевого слова кроме непосредственно BEGIN.

И это ставит интересный вопрос: почему?

Если мы посмотрим на структуры C программы, мы обнаружим, что все функции совсем одинаковы, за исключением того, что основная программа идентифицируется по своему имени "main". Так как функции C могут появляться в любом порядке, основная программа так же может быть в любом месте модуля компиляции.

В Паскале наоборот, все переменные и процедуры должны быть объявлены прежде чем они используются, что означает, что нет никакого смысла помещать что-либо после основной программы... к ней никогда нельзя будет обратиться. "Основная программа" не идентифицирована вообще, кроме того, что эта часть кода следует после глобального BEGIN. Другими словами если это не что-нибудь еще, это должна быть основная программа.

Это приводит к немалой путанице для начинающих программистов, а для больших программ на Паскале иногда вообще трудно найти начало основной программы. Это ведет к соглашениям типа идентификации ее в комментариях:

```
BEGIN { of MAIN }
```

Это всегда казалось мне немного клуджем. Возникает вопрос: Почему обработка основной программы должна так отличаться от обработки процедур? Теперь, когда мы осознали, что объявление процедур это просто... часть глобальных объявлений... не является ли основная программа также просто еще одним объявлением?

Ответ - да, и обрабатывая ее таким способом мы можем упростить код и сделать его значительно более ортогональным. Я предлагаю использовать для идентификации основной программы явное ключевое слово PROGRAM (Заметьте, что это означает, что мы не можем начать с него файл, как в Паскале). В этом случае наша БНФ становится:

```
<declaration> ::= <data decl> | <procedure> | <main program>

<procedure> ::= PROCEDURE <ident> <begin-block>

<main program> ::= PROGRAM <ident> <begin-block>
```

Код также смотрится намного лучше, по крайней мере в том смысле, что DoMain и DoProc выглядят более похоже:

```
-- --------------------------------------------------------------
-- Parse and translate a main program
: doMain ( -- )
        'P' match
```

```
        getname prog$ char-place fin
        prolog
        beginblock ;
-- ---------------------------------------------------------------
.
.
.
-- ---------------------------------------------------------------
-- Parse and translate global declarations
: topdecls ( -- )
        BEGIN  Look '.' <>
        WHILE  CASE Look
                'v' OF Decl   ENDOF
                'p' OF DoProc ENDOF
                'P' OF DoMain ENDOF
                    S" Unrecognized keyword' Look name.aborts
                ENDCASE
                fin
        REPEAT ;
-- ---------------------------------------------------------------
-- Main Program
: TINY ( -- ) init topdecls epilog ;
-- ---------------------------------------------------------------
```

Так как объявление основной программы теперь внутри цикла TopDecl, возникают некоторые трудности. Как мы можем гарантировать, что она - последняя в файле? И выйдем ли мы когда либо из цикла? Мой ответ на второй вопрос, как вы можете видеть, - в том, чтобы вернуть нашего старого друга точку. Как только синтаксический анализатор увидит ее дело сделано(calls3.frt) .

Ответ на первый вопрос: он зависит от того, насколько мы хотим защищать программиста от глупых ошибок. В коде, который я показал, нет ничего, предохраняющего программиста от добавления кода после основной программы... даже другой основной программы. Код просто не будет доступен. Однако, мы могли бы обращаться к нему через утверждение FORWARD, которое мы предоставим позже. Фактически, многие программисты на ассемблере любят использовать область сразу после программы для объявления больших, неинициализированных блоков данных, так что действительно может быть некоторый смысл не требовать, чтобы основная программа была последней. Мы оставим все как есть.

Если мы решим, что должны дать программисту немного больше помощи чем сейчас, довольно просто добавить некоторую логику, которая выбросит нас из цикла как только основная программа будет обработана. Или мы могли бы по крайней мере сообщать об ошибке если кто-то попытается вставить две основных.

ВЫЗОВ ПРОЦЕДУРЫ

Если вы удовлетворены работой программы, давайте обратимся ко второй половине уравнения... вызову.

Рассмотрим БНФ для вызова процедуры:

```
<proc_call> ::= <identifier>
```

с другой стороны БНФ для операции присваивания:

```
<assignment> ::= <identifier> '=' <expression>
```

Кажется у нас проблема. Оба БНФ утверждения с правой стороны начинаются с токена <identifier>. Как мы предполагаем узнать, когда мы видим идентификатор, имеем ли мы вызов процедуры или операцию присваивания? Это похоже на случай, когда наш синтаксический анализатор перестает быть предсказывающим и действительно это точно такой случай. Однако, оказывается эту проблему легко решить, так как все, что мы должны

сделать - посмотреть на тип идентификатора записанный в таблице идентификаторов. Как мы обнаружили раньше, небольшое локальное нарушение правила предсказывающего синтаксического анализа может быть легко обработано как специальный случай.

Вот как это делается:

```
-- ---------------------------------------------------------------
-- Parse and translate an assignment statement
: assignment ( char -- )
        >R
          '=' match expression
        R> storevariable ;
-- ---------------------------------------------------------------
-- Decide if a statement is an assignment or procedure call
: assignorproc ( -- )
        getname >R
        CASE R@ typeof
           0  OF R> .undefined  ENDOF
          'v' OF R> assignment  ENDOF
          'p' OF R> callproc    ENDOF
                 S" Identifier " R> CHAR-APPEND
                 S" cannot be used here" $+ aborts
        ENDCASE ;
-- ---------------------------------------------------------------
-- Parse and translate a block of statements
: doBlock ( -- )
        BEGIN  Look 'e' <>
        WHILE  assignorproc
               fin
        REPEAT ;
-- ---------------------------------------------------------------
```

Как вы можете видеть, процедура Block сейчас вызывает AssignOrProc вместо Assignment. Назначение этой новой процедуры просто считать идентификатор, определить его тип и затем вызвать процедуру, соответствующую этому типу. Так как имя уже прочитано, мы должны передать его в эти две процедуры и соответственно изменить Assignment. Процедура CallProc - это просто подпрограмма генерации кода:

```
-- -----------------------------------------------------------------------
-- Call a procedure
: callproc ( char -- ) S"  offset NEAR call, " ROT CHAR-PREPEND emitln ;
-- -----------------------------------------------------------------------
```

Хорошо, к этому моменту у нас есть компилятор, который может работать с процедурами.(calls4.frt) Стоить отметить, что процедуры могут вызывать процедуры с любой степенью вложенности. Так что, даже хотя мы и не разрешаем вложенные объявления, нет ничего, чтобы удерживало нас от вложенных вызовов, точно так, как мы ожидали бы на любом языке. Мы получили это и это было не слишом сложно, не так ли?

Конечно, пока мы можем работать только с процедурами, которые не имеют параметров. Процедуры могут оперировать глобальными переменными по их глобальным именам. Так что к этому моменту мы имеем эквивалент конструкции Бейсика GOSUB. Не слишком плохо... в конце концов масса серьезных программ была написана с применением GOSUBа., но мы можем добиться большего и добьемся. Это следующий шаг.

ПЕРЕДАЧА ПАРАМЕТРОВ

Снова, все мы знаем основную идею передачи параметров, но давайте просто для надежности разберем ее заново.

Вообще, процедуре предоставляется список параметров, например:

```
PROCEDURE FOO(X, Y, Z)
```

В объявлении процедуры параметры называются формальными параметрами и могут упоминаться в теле процедуры по своим именам. Имена, используемые для формальных параметров в действительности произвольны. Учитывается только позиция. В примере выше имя 'X' просто означает "первый параметр" везде, где он используется.

Когда процедура вызывается, "фактические параметры" переданные ей, связаны с формальными параметрами на взаимно-однозначном принципе.

БНФ для синтаксиса выглядит приблизительно так:

```
<procedure> ::= PROCEDURE <ident>
               '(' <param-list> ')' <begin-block>


<param_list> ::= <parameter> ( ',' <parameter> )* | null
```

Аналогично, вызов процедуры выглядит так:

```
<proc call> ::= <ident> '(' <param-list> ')'
```

Обратите внимание, что здесь уже есть неявное решение, встроенное в синтаксис. Некоторые языки, такие как Pascal и Ada разрешают списку параметров быть необязательным. Если нет никаких параметров, вы просто полностью отбрасываете скобки. Другие языки, типа C и Modula-2, требуют скобок даже если список пустой.   Ясно, что пример, который мы только что привели, соответствует первой точке зрения. Но, сказать правду, я предпочитаю последний. Для одних процедур решение кажется должно быть в пользу "безсписочного" подхода. Оператор

```
initialize; ,
```

стоящий отдельно, может означать только вызов процедуры. В синтаксических анализаторах, которые мы писали, мы преимущественно использовали процедуры без параметров и было бы позором каждый раз заставлять писать пустую пару скобок.

Но позднее мы также собираемся использовать и функции. И так как функции могут появляться в тех же самым местах что и простые скалярные идентификаторы, вы не сможете сказать об их различиях. Вы должны вернуться к объявлениям, чтобы выяснить это. Некоторые люди полагают, что это преимущество. Их аргументы в том, что идентификатор замещается значением и почему вас заботит, сделано ли это с помощъю подстановки или функции? Но нас это иногда заботит, потому что функция может выполняться довольно долго. Если написав простой идентификатор в данном выражении мы можем понести большие затраты во время выполнения, то мне кажется, что мы должны быть осведомлены об этом.

В любом случае, Никлаус Вирт разработал и Pascal и Modula-2. Я оправдаю его и полагаю что он имел веские причины для изменения правил во втором случае!

Само собой разумеется, легко принять любую точку зрения на то, как разрабатывать язык, так что это строго вопрос персонального предпочтения. Делайте это таким способом, какой вам больше нравится.

Перед тем как пойти дальше, давайте изменим транслятор для поддержки списка параметров (возможно пустого). Пока мы не будем генерировать никакого дополнительного кода... просто анализировать синтаксис. Код для обработки объявления имеет ту же самую форму, которую мы видели раньше когда работали со списками переменных:

```
-- ------------------------------------------------------------
-- Process the formal parameter list of a procedure
: formallist ( -- )
        '(' match
        Look ')' <>
           IF   formalparam
```

```
                BEGIN  Look ',' =
                WHILE  ',' match formalparam
                REPEAT
        ENDIF
        ')' match ;
-- -------------------------------------------------------------
```

В процедуру DoProc необходимо добавить строчку для вызова FormalList:

```
-- -------------------------------------------------------------
-- Parse and translate a procedure declaration
: doProc ( -- )
        'p' match
        getname >R
        formallist
        fin
        R@ 'p' addentry
        R> procheader
        beginblock
        _ret ;
-- -------------------------------------------------------------
```

Сейчас код для FormalParam всего лишь пустышка, который просто пропускает имена переменных:

```
-- -------------------------------------------------------------
-- Process a formal parameter
: formalparam ( -- ) getname DROP ;
-- -------------------------------------------------------------
```

Для фактического вызова процедуры должен быть аналогичный код для обработки списка фактических параметров:

```
-- -------------------------------------------------------------
-- Process an actual parameter
: param ( -- ) getname DROP ;
-- -------------------------------------------------------------
-- Process the actual parameter list for a procedure call
: paramlist ( -- )
        '(' match
        Look ')' <>
           IF   param
                BEGIN Look ',' =
                WHILE ',' match param
                REPEAT
        ENDIF
        ')' match ;
-- -------------------------------------------------------------
-- Process a procedure call
: callproc ( char -- )
        paramlist _call ;
-- -------------------------------------------------------------
```

Обратите внимание, что CallProc больше не является просто простой подпрограммой генерации кода. Она имеет некоторую структуру. Для обработки я переименовал подпрограмму генерации кода в просто Call и вызвал ее из CallProc.

Итак, если вы добавите весь этот код в ваш транслятор и протестируете его(calls5.frt), , вы обнаружите, что действительно можете правильно анализировать синтаксис. Обращаю ваше внимание на то, что здесь нет никакой проверки того, что количество (и, позднее, тип) формальных и фактических параметров совпадает. В промышленном компиляторе, мы конечно должны делать это. Сейчас мы игнорируем эту проблему той причине, что структура нашей таблицы идентификаторов пока не дает нам места для сохранения необходимой

информации. Позднее мы подготовим место для этих данных и тогда сможем работать с этой проблемой.

СЕМАНТИКА ПАРАМЕТРОВ

До этого мы имели дело с синтаксисом передачи параметров и получили механизм синтаксического анализа для его обработки. Сейчас мы должны рассмотреть семантику, т.е. действия, которые должны быть предприняты когда мы столкнемся с параметрами. Это ставит нас перед вопросом выбора способа передачи параметров.

Существует более чем один способ передачи параметров и способ, которым мы сделаем это, может иметь глубокое влияние на характер языка. Так что это одна из тех областей, где я не могу просто дать вам свое решение. Скорее, было бы важно чтобы мы потратили некоторое время на рассмотрение альтернатив, так чтобы вы могли, если захотите, пойти своим путем.

Есть два основных способа передачи параметров:

- По значению
- По ссылке (адресу)

Различия лучше всего видны в свете небольшого исторического обзора.

Старые компиляторы Фортрана передавали все параметры по ссылке. Другими словами, фактически передавался адрес параметра. Это означало, что вызываемая подпрограмма была вольна и считывать и изменять этот параметр, что часто и происходило, как будто это была просто глобальная переменная. Это был фактически самый эффективный способ и он был довольно простым, так как тот же самый механизм использовался во всех случаях с одним исключением, которое я кратко затрону.

Хотя имелись и проблемы. Многие люди чувствовали, что этот метод создавал слишком большую связь между вызванной и вызывающей подпрограммой. Фактически, это давало подпрограмме полный доступ ко всем переменным, которые появлялись в списке параметров.

Часто нам не хотелось бы фактически изменять параметр а только использовать его как входные данные. К примеру, мы могли бы передавать счетчик элементов в подпрограмму и хотели бы затем использовать этот счетчик в цикле DO. Во избежание изменения значения в вызываемой программе мы должны были сделать локальную копию входного параметра и оперировать только его копией. Некоторые программисты на Фортране фактически сделали практикой копирование всех параметров, исключая те, которые должны были использоваться как возвращаемые значения. Само собой разумеется, все это копирование победило добрую часть эффективности, связанной с этим методом.

Существовала, однако, еще более коварная проблема, которая была в действительности не просто ошибкой соглашения "передача по ссылке", а плохой сходимостью нескольких решений реализации.

Предположим, у нас есть подпрограмма:

```
SUBROUTINE FOO(X, Y, N)
```

где N - какой-то входной счетчик или флажок. Часто нам бы хотелось иметь возможность передавать литерал или даже выражение вместо переменной, как например:

```
CALL FOO(A, B, J + 1)
```

Третий параметр не является переменной, и поэтому он не имеет никакого адреса. Самые ранние компиляторы Фортрана не позволяли таких вещей, так что мы должны были прибегать к ухищрениям типа:

```
K = J + 1
```

```
CALL FOO(A, B, K)
```

Здесь снова требовалось копирование и это бремя ложилось на программистов. Не хорошо.

Более поздние реализации Фортрана избавились от этого, разрешив использовать выражения как параметры. Что они делали - назначали сгенерированную компилятором переменную, сохраняли значение выражения в этой переменной и затем предавали адрес выражения.

Пока все хорошо. Даже если подпрограмма ошибочно изменила значение анонимной переменной, кто об этом знал или кого это заботило? При следующем вызове она в любом случае была бы рассчитана повторно.

Проблема возникла когда кто-то решил сделать вещи более эффективными. Они рассуждали, достаточно справедливо, что наиболее общим видом "выражений" было одиночное целочисленное значение, как в:

```
CALL FOO(A, B, 4)
```

Казалось неэффективным подходить к проблеме "вычисления" такого целого числа и сохранять его во временной переменной только для передачи через список параметров. Так как мы в любом случае передавали адрес, казалось имелся большой смысл в том, чтобы просто передавать адрес целочисленного литерала, 4 в примере выше.

Чтобы сделать вопрос более интересным большинство компиляторов тогда и сейчас идентифицирует все литералы и сохраняет их отдельно в "литерном пуле", так что мы должны сохранять только одно значение для каждого уникального литерала. Такая комбинация проектных решений: передача выражений, оптимизация литералов как специальных случаев и использование литерного пула - это то, что вело к бедствию.

Чтобы увидеть, как это работает, вообразите, что мы вызываем подпрограмму FOO как в примере выше, передавая ей литерал 4. Фактически, что передается - это адрес литерала 4, который сохранен в литерном пуле. Этот адрес соответствует формальному параметру K в самой подпрограмме.

Теперь предположите, что без ведома программиста подпрограмма FOO фактически присваивает K значение -7. Неожиданно, литерал 4 в литерном пуле меняется на -7. В дальнейшем, каждое выражение, использующее 4, и каждая подпрограмма, в которую передают 4, будут использовать вместо этого значение -7! Само собой разумеется, что это может привести к несколько причудливому и труднообъяснимому поведению. Все это дало концепции передачи по ссылке плохое имя, хотя, как мы видели, в действительности это была комбинация проектных решений, ведущая к проблеме.

Несмотря на проблему, подход Фортрана имел свои положительные моменты. Главный из них - тот факт, что мы не должны поддерживать множество механизмов. Та же самая схема передачи адреса аргумента работает для всех случаев, включая массивы. Так что размер компилятора может быть сокращен.

Частично из-за этого подводного камня Фортрана и частично просто из-за уменьшенной связи, современные языки типа C, Pascal, Ada и Modula 2 в основном передают скаляры по значению.

Это означает, что значение скаляра копируется в отдельное значение, используемое только для вызова. Так как передаваемое значение - копия, вызываемая процедура может использовать его как локальную переменную и изменять ее любым способом, каким нравится. Значение в вызывающей программе не будет изменено.

Сначала может показаться, что это немного неэффективно из-за необходимости копировать параметр. Но запомните, что мы в любом случае окажемся перед необходимостью выбирать какое-то значение, является ли оно непосредственно параметром или его адресом.

Внутри подпрограммы, использование передачи по значению определенно более эффективно, так как мы устраняем один уровень косвенности. Наконец, мы видели раньше, что в Фортране часто было необходимо в любом случае делать копии внутри подпрограммы, так что передача по значению уменьшает количество локальных переменных. В целом, передача по значению лучше.

Исключая одну маленькую деталь: если все параметры передаются по значению, у вызванной процедуры нет никакого способа возвратить результат в вызвавшую! Переданный параметр не изменяется в вызвавшей подпрограмме а только в вызванной. Ясно, что так работы не сделать.

Существуют два эквивалентных ответа на эту проблему. В Паскале Вирт предусмотрел параметры-переменные, которые передаются по ссылке. VAR параметр не что иное как наш старый друг параметр Фортрана с новым именем и расцветкой для маскировки. Вирт аккуратно обходит проблему "изменения литерала" так же как проблему "адрес выражения" с помощью простого средства, разрешая использовать в качестве фактических параметров только переменные. Другими словами, это тоже самое ограничение, которое накладывали самые ранние версии Фортрана.

Си делает ту же самую вещъ, но явно. В C все параметры передаются по значению. Однако одним из видов переменных, которые поддерживает C, является указатель. Так передавая указатель по значению, вы в действительности передаете то, на что он указывает по ссылке. В некоторых случаях это работает даже еще лучше, потому что даже хотя вы и можете изменить указываемую переменную на все, что хотите, вы все же не сможете изменить сам указатель. В функции типа strcpy, к примеру, где указатель увеличивается при копировании строки, мы в действительности увеличиваем только копии указателей, так что значение указателей в вызвавшей процедуре все еще остается каким было. Чтобы изменить указатель вы должны передавать указатель на указатель.

Так как мы просто проводим эксперименты, мы рассмотрим и передачу по значению и передачу по ссылке. Таким образом у нас будет возможность использовать любой из них как нам нужно. Стоит упомянуть, что было бы тяжело использовать здесь подход C, так как указатель это другой тип а типы мы еще не изучали!

ПЕРЕДАЧА ПО ЗНАЧЕНИЮ

Давайте просто попробуем некоторые нехитрые вещи и посмотрим, куда они нас приведут. Давайте начнем со случая передачи по значению. Рассмотрим вызов процедуры:

```
FOO(X, Y)
```

Почти единственным приемлемым способом передачи данных является передача через стек ЦПУ. Поэтому, код который мы бы хотели видеть сгенерированным мог бы выглядеть так:

```
X dword-ptr  push,       ; Push X
Y dword-ptr  push,       ; Push Y
FOO offset NEAR  call,   ; Call FOO
```

Это конечно не выглядит слишком сложным!

Когда BSR выполнен центральный процессор помещает адрес возврата в стек и переходит к FOO. В этой точке стек будет выглядеть следующим образом:

```
          .
       .

          Value of X     (4 bytes)
          Value of Y     (4 bytes)
          return address (4 bytes)
ESP -->   [ ..... ]
```

Так что значения параметров имеют адреса с фиксированными смещениями от указателя стека. В этом примере адреса такие:

```
X:  8(SP)
Y:  4(SP)
```

Теперь рассмотрим, на что могла бы походить вызываемая процедура:

```
PROCEDURE FOO(A, B)
BEGIN
     A = B
END
```

(Помните, что имена формальных параметров произвольные... учитываются только позиции).

Желаемый код мог бы выглядеть так:

```
FOO: [esp 4 +] -> eax  mov,
     eax -> [esp 8 +]  mov,
     ret,
```

Обратите внимание, что для адресации формальных параметров нам будет необходимо знать, какую позицию они занимают в списке параметров. Это подразумевает некоторые изменения в содержимом таблицы идентификаторов. Фактически, в нашем односимвольном случае лучше всего просто создать новую таблицу идентификаторов для формальных параметров.

Давайте начнем с объявления новой таблицы:

```
CREATE params 26 CHARS ALLOT
```

Нам также необходимо отслеживать, сколько параметров имеет данная процедура:

```
0 VALUE #params
```

И мы должны инициализировать новую таблицу. Теперь, не забудьте, что список формальных параметров будет различным для каждой процедуры, которые мы обрабатываем, так что мы будем должны инициализировать эту таблицу заново для каждой процедуры. Вот инициализатор:

```
-- ------------------------------------------------------------
-- Initialize parameter table to null
: clearparams ( -- )
        params 26 CHARS ERASE
        CLEAR #params ;
-- ------------------------------------------------------------
```

Мы поместим обращение к этой процедуре в Init и также в конец DoProc:

```
-- ------------------------------------------------------------
-- Initialize
: init ( -- )
        CLEAR Lcount
        clearST
        clearparams
        CR getchar skipwhite ;
-- ------------------------------------------------------------
.
.
.
-- ------------------------------------------------------------
-- Parse and translate a procedure declaration
: doProc ( -- )
```

```
        'p' match
        getname >R
        formallist
        fin
        R@ 'p' addentry
        R> procheader
        beginblock
        _ret
        clearparams ;
-- --------------------------------------------------------------
```

Обратите внимание, что вызов внутри DoProc гарантирует, что таблица будет чиста, когда мы в основной программе.

Хорошо, теперь нам нужны несколько процедур для работы с таблицей. Следующие несколько функций являются по существу копиями InTable, TypeOf и т.д.:

```
-- --------------------------------------------------------------
-- Find the parameter number
: paramnumber ( char -- u ) 'A' - params + C@ ;
-- --------------------------------------------------------------
-- See if an identifier is a parameter
: param? ( char -- tf ) 'A' - params + C@ 0<> ;
-- --------------------------------------------------------------
-- Add a new parameter to table
: addparam ( char -- )
        DUP param? IF  .duplicate  ENDIF
        1 +TO #params
        #params SWAP 'A' - params + C! ;
-- --------------------------------------------------------------
```

Наконец, нам понадобятся некоторые подпрограммы генерации кода:

```
-- --------------------------------------------------------------
-- Load a parameter to the primary register
: loadparameter ( index -- )
        #params SWAP - 1+ CELLS ( offset ) >R
        S" [esp "  R> (.) $+  S"  +] -> eax  mov," $+ emitln ;
-- --------------------------------------------------------------
-- Store a parameter from the primary register
: storeparameter ( index -- )
        #params SWAP - 1+ CELLS ( offset ) >R
        S" eax -> [esp "  R> (.) $+  S"  +]  mov," $+ emitln ;
-- --------------------------------------------------------------
-- Push the primary register to the stack
: _push ( -- )  S" eax  push," emitln ;
-- --------------------------------------------------------------
```

(Последнюю подпрограмму мы уже видели прежде, но ее не было в этой остаточной версии программы.)

После этих приготовлений мы готовы работать с семантикой процедур со списками вызовов (помните, что код для работы с синтаксисом уже на месте).

Давайте начнем с обработки формальных параметров. Все что мы должны сделать - добавить каждый параметр в таблицу идентификаторов параметров:

```
-- --------------------------------------------------------------
-- Process a formal parameter
: formalparam ( -- ) getname addparam ;
-- --------------------------------------------------------------
```

Теперь, что делать с формальными параметрами, когда они появляются в теле процедуры? Это требует немного больше работы. Мы должны сначала определить, что это формальный параметр. Чтобы сделать это, я написал модифицированную версию TypeOf:

```
-- ------------------------------------------------------------------
-- Get type of symbol
: typeof ( char -- ) DUP param? IF  DROP 'f'  ELSE  ST[n] C@  ENDIF ;
-- ------------------------------------------------------------------
```

(Обратите внимание, что так как TypeOf теперь вызывает IsParam, возможно будет необходимо изменить ее местоположение в программе.)

Мы также должны изменить AssignOrProc для работы с этим новым типом:

```
-- ------------------------------------------------------------
-- Decide if a statement is an assignment or procedure call
: assignorproc ( -- )
        getname >R
        CASE R@ typeof
           0  OF R> .undefined  ENDOF
          'v' OF R> assignment  ENDOF
          'f' OF R> assignment  ENDOF
          'p' OF R> callproc    ENDOF
              S" Identifier `" R> CHAR-APPEND
              S" ' cannot be used here" $+ aborts
        ENDCASE ;
-- ------------------------------------------------------------
```

Наконец, код для обработки операции присваивания и выражения должен быть расширен:

```
-- ------------------------------------------------------------
-- Parse and translate an expression
-- Vestigial version
        getname
        DUP param? IF  paramnumber loadparameter
                 ELSE  loadvariable
                ENDIF ;
-- ------------------------------------------------------------
-- Parse and translate an assignment statement
: assignment ( char -- )
        >R  '=' match expression  R>
        DUP param? IF  paramnumber storeparameter
                 ELSE  storevariable
                ENDIF ;
-- ------------------------------------------------------------
```

Как вы можете видеть, эти процедуры обработают каждое встретившееся имя переменной или как формальный параметр или как глобальную переменную, в зависимости от того, появляется ли оно в таблице идентификаторов параметров. Запомните, что мы используем только остаточную форму Expression. В конечной программе изменения, показанные здесь, должны быть добавлены в Factor а не Expression.

Осталось самое простое. Мы должны только добавить семантику в фактический вызов процедуры, что мы можем сделать с помощью одной новой строки кода:

```
-- ------------------------------------------------------------
-- Process an actual parameter
: param ( -- ) expression _push ;
-- ------------------------------------------------------------
```

Так вот. Добавьте эти изменения в вашу программу и испытайте (calls6.frt)ее. Попробуйте объявить одну или две процедуры, каждая со списком формальных параметров. Затем сделайте какие-нибудь присваивания, используя комбинации глобальных и формальных

параметров. Вы можете вызывать одну процедуру из другой, но вы не можете объявлять вложенные процедуры. Вы можете даже передавать формальные параметры из одной процедуры в другую. Если бы мы имели здесь полный синтаксис языка, вы могли бы также читать и выводить формальные параметры или использовать их в сложных выражениях.

ЧТО НЕПРАВИЛЬНО?

Тут вы могли бы подумать: Уверен, здесь должно быть что-то большее чем несколько сохранений и восстановлений из стека. Для передачи параметров здесь должно быть что-то большее чем тут есть.

Вы были бы правы. Фактически, код, который мы здесь генерируем, оставляет желать лучшего в нескольких случаях.

Самая явная оплошность в том, что он неправильный! Если вы оглянетесь на код для вызова процедур, вы увидите, что вызывающая подпрограмма помещает каждый фактический параметр в стек перед тем, как она вызывает процедуру. Процедура использует эту информацию, но она не изменяет указатель стека. Это означает, что содержимое все еще остается там когда мы возвращаемся. Кто-то должен очистить стек или мы скоро окажемся в очень трудной ситуации!

К счастью, это легко исправить. Все, что мы должны сделать - это увеличить указатель стека когда мы закончим.

Должны ли мы делать это в вызывающей программе или в вызываемой процедуре? Некоторые люди позволяют вызываемой процедуре очищать стек, так как требуется генерировать меньше кода на вызов и так как процедура, в конце концов, знает сколько параметров она получила. Но это означает, что она должна что-то делать с адресом возврата чтобы не потерять его.

Я предпочитаю разрешить очистку в вызывающей программе, так что вызываемая процедура должна только выполнить возврат. Также это кажется немного более сбалансированным так как именно вызывающая программа первой "засорила" стек. Но это означает, что вызывающая программа должна запоминать сколько элементов помещено в стек. Чтобы сделать проще, я изменил процедуру ParamList на функцию, возвращающую количество помещенных байт:

```
-- -------------------------------------------------------------
-- Process the parameter list for a procedure  call
: paramlist ( -- #bytes )
        0 LOCAL N
        '(' match
        Look ')' <>
           IF   param 1 +TO N
                BEGIN Look ',' =
                WHILE ',' match param 1 +TO N
                REPEAT
        ENDIF
        ')' match
        N CELLS ;
-- -------------------------------------------------------------
```

Процедура CallProc затем использует его для очистки стека:

```
-- -------------------------------------------------------------
-- Process a procedure call
: callproc ( char -- ) paramlist >R _call R> _cleanstack ;
-- -------------------------------------------------------------
```

Здесь я создал еще одну подпрограмму генерации кода:

```
-- ---------------------------------------------------------------
-- Adjust the stack pointer upwards by n bytes
: _cleanstack ( #bytes -- )
        ?DUP IF   (.) S"  d# -> esp  add," $+ emitln  ENDIF ;
-- ---------------------------------------------------------------
```

ОК, если вы добавили этот код в ваш компилятор, я думаю вы убедитесь, что стек теперь под контролем.(calls7.frt)

Следующая проблема имеет отношение к нашему способу адресации относительно указателя стека. Это работает отлично на наших простых примерах, так как с нашей элементарной формой выражений никто больше не засоряет стек. Но рассмотрим другой пример, такой простой как:

```
PROCEDURE FOO(A, B)
BEGIN
     A = A + B
END
```

Код, сгенерированный нехитрым синтаксическим анализатором, мог бы быть:

```
FOO: [esp 8 +] -> eax mov, ; fetch a
     eax push,             ; push it
     [esp 4 +] -> eax mov, ; fetch b
     [esp]     -> eax add, ; add a
     eax -> [esp 8 +] mov, ; store a
     ret,
```

Это было бы неправильно. Когда мы помещаем первый аргумент в стек, смещения для двух формальных параметров больше не 4 и 6, я 6 и 8. Поэтому вторая выборка вернула бы снова A а не B.

Но это не конец света. Я думаю, вы можете видеть, что все, что мы должны делать - изменять смещение каждый раз, когда мы помещаем в стек и что фактически и делается если ЦПУ не имеет поддержки других методов.

К счастью, все-же, 68000 имеет такую поддержку. Поняв, что этот ЦПУ мог бы использоваться со многими компиляторами языков высокого уровня, Motorola решила добавить прямую поддержку таких вещей.

Проблема, как вы можете видеть в том, что когда процедура выполняется, указатель стека скачет вверх и вниз, и поэтому использование его как ссылки для доступа к формальным параметрам становится неудобным. Решение состоит в том, чтобы вместо него определить и использовать какой-то другой регистр. Этот регистр обычно устанавливается равным подлинному указателю стека и называется указателем кадра.

Команда LINK из набора инструкций 68000 позволяет вам объявить такой указатель кадра и установить его равным указателю стека и все это в одной команде. Фактически, она делает даже больше чем это. Так как этот регистр может использоваться для чего-то еще в вызывающей процедуре, LINK также помещает текущее значение регистра в стек. Вы можете также добавить значение к указателю стека чтобы создать место для локальных переменных.

В дополнение к LINK есть UNLK, которая просто восстанавливает указатель стека и выталкивает старое значение обратно в регистр.

С использованием этих двух команд код для предыдущего примера станет:

```
FOO: 0 0 enter,
     [ebp 12 +] -> eax  mov, ; fetch a
     eax push,               ; push it
     [ebp  8 +] -> eax  mov, ; fetch b
     [esp] -> eax  add,      ; add a
     eax -> [ebp 12 +]  mov, ; store a
```

```
        leave,
        ret,
```

Исправить компилятор для генерации этого кода намного проще чем объяснить. Все, что нам нужно сделать - изменить генерацию кода в DoProc. Так как из-за этого код становится немного больше одной строки, я создал новые процедуры, схожие с процедурами Prolog и Epilog, вызываемыми DoMain:

```
-- ---------------------------------------------------------------
-- Write the prolog for a procedure
: procprolog ( char -- )
        S" SUBROUTINE " ROT CHAR-APPEND emitdln
        S" 0 0  enter," emitln ;
-- ---------------------------------------------------------------
-- Write the epilog for a procedure
: procepilog ( -- )
        S" leave," emitln
        S" ret,"   emitln ;
-- ---------------------------------------------------------------
```

Процедура DoProc теперь просто вызывает их:

```
-- ---------------------------------------------------------------
-- Parse and translate a procedure declaration
: doProc ( -- )
        'p' match
        getname >R
        formallist
        fin
        R@ 'p' addentry
        R> procprolog
        beginblock
        procepilog
        clearparams ;
-- ---------------------------------------------------------------
```

В заключение, мы должны изменить ссылки на SP в процедурах LoadParam и StoreParam:

```
-- ---------------------------------------------------------------
-- Load a parameter to the primary register
: loadparameter ( index -- )
        #params SWAP - 2+ CELLS ( offset ) >R
        S" [ebp "  R> (.) $+  S"  +] -> eax  mov," $+ emitln ;
-- ---------------------------------------------------------------
-- Store a parameter from the primary register
: storeparameter ( index -- )
        #params SWAP - 2+ CELLS ( offset ) >R
        S" eax -> [ebp "  R> (.) $+  S"  +]  mov," $+ emitln ;
-- ---------------------------------------------------------------
```

(Заметьте, что вычисление Offset изменяется чтобы учесть дополнительное сохранение A6.)

Это все что требуется. Попробуйте и посмотрите как вам это нравится(calls8.frt.

К этому моменту мы генерируем некоторый относительно хороший код для процедур и вызовов процедур. С ограничениями, что нет никаких локальных переменных (пока) и не разрешено вложение процедур этот код именно то что нам нужно.

Все еще остается только одна небольшая проблема:

У нас нет способа возвратить результат в вызывающую программу!

Но это, конечно, не ограничение генерируемого нами кода, а ограничение, свойственное протоколу передачи по значению. Обратите внимание, что мы можем использовать

формальные параметры любым способом внутри процедуры. Мы можем вычислять для них новое значение, использовать их как счетчики циклов (если бы мы имели циклы!) и т.д. Так что код делает то, что предполагается. Чтобы решить эту последнюю проблему мы должны рассмотреть альтернативный протокол.

ПЕРЕДАЧА ПО ССЫЛКЕ

Это просто теперь, когда мы уже имеем механизм. Мы только должны внести несколько изменений в генерацию кода. Вместо помещения значения в стек, мы должны помещать адрес. Оказывается, 68000 имеет инструкцию PEA которая как раз делает это.

Для этого мы сделаем новую версию тестовой программы. Перед тем, как сделать что-нибудь еще, сделайте копию программы в ее текущем состоянии, потому что позже она понадобится нам снова.

Давайте начнем с рассмотрения кода, который мы хотели бы видеть сгенерированным для нового случая. Используя тот же самый пример что и раньше, мы должны вызов

```
FOO(X, Y)
```

оттранслировать в:

```
X d# push,             ; Push the address of X
Y d# push,             ; Push the address of Y
FOO offset NEAR  call, ; Call FOO
```

Это просто вопрос небольших изменений в Param:

```
-- ------------------------------------------------------------
-- Process an actual parameter
: param ( -- ) S"  d# push," getname CHAR-PREPEND emitln ;
-- ------------------------------------------------------------
```

(Обратите внимание, что при передачей по ссылке мы не можем использовать выражения в списке параметров, поэтому Param может просто непосредственно считывать имя).

На другой стороне, ссылки на формальные параметры должны получить один уровень косвенности:

```
FOO: 0 0 enter,
     [ebp 12 +] -> eax  mov, ; fetch address of A
     [eax] -> eax mov,       ; fetch A
     eax push,               ; push it
     [ebp  8 +] -> eax  mov, ; fetch address of B
     [eax] -> eax mov,       ; fetch B
     [esp] -> eax add,
     [ebp 12 +] -> ebx  mov, ; get address of A
     eax -> [ebx]       mov, ; store A
     leave,
     ret,
```

Все это может быть обработано с изменениями в LoadParam and StoreParam:

```
-- ------------------------------------------------------------
-- Load a parameter to the primary register
: loadparameter ( index -- )
        #params SWAP - 2+ CELLS ( offset ) >R
        S" [ebp "  R> (.) $+  S"  +] -> ebx  mov," $+ emitln
        S" [ebx] -> eax  mov," emitln ;
-- ------------------------------------------------------------
-- Store a parameter from the primary register
: storeparameter ( index -- )
        #params SWAP - 2+ CELLS ( offset ) >R
        S" [ebp "  R> (.) $+  S"  +] -> ebx  mov," $+ emitln
```

```
      S" eax -> [ebx]  mov," emitln ;
-- -----------------------------------------------------------
```

Для правильного расчета, мы также должны изменить одну строку в ParamList:

ParamList := 4 * N;

Теперь должно работать. Испытайте компилятор и посмотрите, генерирует ли он приемлемый код. Как вы увидите, код вряд ли оптимален, так как мы перезагружаем регистр адреса каждый раз, когда необходим параметр. Но это соответствует нашему принципу KISS - просто генерировать код который работает. Мы только сделаем здесь небольшое замечание, что есть еще один кандидат для оптимизации и пойдем дальше.

Теперь мы научились обрабатывать параметры использую передачу по значению и передачу по ссылке. В реальном мире, конечно, мы хотели бы иметь возможность работать с обоими методами. Однако пока мы не можем этого сделать, потому что у нас еще не было урока по типам.

Если мы можем иметь только один метод, то, конечно, это должен быть старый добрый Фортранов метод передачи по ссылке, так как это единственный способ, которым процедуры могут возвращать значения в вызвавшую программу.

Это, фактически, будет одним из различий между TINY и KISS. В следующей версии TINY мы будем использовать передачу по ссылке для всех параметров. KISS будет поддерживать оба метода.

ЛОКАЛЬНЫЕ ПЕРЕМЕННЫЕ

Пока мы не сказали ничего о локальных переменных и наше определение процедур не разрешает их. Само собой разумеется, что это большой пробел в нашем языке и он должен быть исправлен.

И снова здесь мы стоим перед выбором: статическое или динамическое хранение?

В старых FORTRAN программах локальные переменные использовали статическое хранение подобно глобальным. То есть, каждая локальная переменная получала имя и распределенный адрес как любая другая переменная и к ней обращались по этому имени.

Нам это легко сделать, используя уже имеющийся механизм распределения. Помните, однако, что локальные переменные могут иметь те же самые имена, что и глобальные переменные. Мы так или иначе должны согласиться с этим, назначая уникальные имена для этих переменных.

Характерная особенность статического хранения в том, что данные выживают при вызове процедуры и возврате. Когда процедура вызывается снова, данные все еще будут здесь. Это может быть преимуществом в некоторых приложениях. Во времена FORTRAN мы применяли такой прием как инициализация флажка, чтобы вы могли сказать когда мы входим в процедуру первый раз и могли бы выполнить любую первоначальную инициализацию, которую необходимо выполнить.

Конечно, эта же "особенность" статического хранения делает рекурсию невозможной. Любое новое обращение к процедуре перепишет данные уже находящиеся в локальных переменных.

Альтернативой является динамическое хранение, при котором память распределяется в стеке точно также как и для переданных параметров. Для это мы уже имеем готовый механизм. Фактически, те же самые подпрограммы, которые работают с переданными (по значению) параметрами в стеке, могут так же легко работать и с локальными переменными... генерируемый код тот же самый. Назначение смещения в инструкции 68000 LINK сейчас такое: мы можем использовать его для регулировки указателя стека при выделении места для локальных переменных. Динамическое хранение, конечно, по существу поддерживает рекурсию.

Когда я впервые начал планировать TINY, я должен признаться имел предубеждение в пользу статического хранения. Просто потому, что старые FORTRAN программы были чрезвычайно эффективны... ранние компиляторы FORTRAN производили качественный код, который и сейчас редко сопоставим с современными компиляторами. Даже сегодня данная программа, написанная на FORTRAN вероятно превзойдет ту же самую программу написанную на C или Pascal, иногда с большим отрывом. (Вот так! Что вы скажете на это заявление!)

Я всегда полагал, что причина имела отношение к двум основным различиям между реализациями Фортрана и другими языками: статическое хранение и передача по ссылке. Я знаю, что динамическое хранение поддерживает рекурсию, но мне всегда казалось немного странным желание мириться с более медленным кодом, который в 95% случаев не нуждается в рекурсии, только для того чтобы получить эту возможность когда она понадобится. Идея состоит в том, что со статическим хранением вы можете использовать не косвенную а абсолютную адресацию, которая должна привести к более быстрому коду.

Позднее, однако, некоторые люди указали мне, что в действительности нет никаких падений производительности связанной с динамическим хранением. Для 68000, к примеру, вы в любом случае не должны использовать абсолютную адресацию... большинство операционных систем требуют переместимый код. И команда 68000

```
[ebp 8 +] -> eax mov,
```

имеет тоже самое время выполнения, что и

```
X dword-ptr -> eax mov, .
```

Так что теперь я убежден, что нет никакой важной причины не использовать динамическое хранение.

Так как такое использование локальных переменных так хорошо соответствует схеме передачи параметров по значению, мы будем использовать эту версию транслятора для иллюстрации (я надеюсь вы сохранили копию!).

Основная идея состоит в том, чтобы отслеживать количество локальных параметров. Затем мы используем это число в инструкции LINK для корректировки указателя стека при выделения для них места. Формальные параметры адресуются как положительные смещения от указателя кадра а локальные как отрицательные смещения. С небольшой доработкой те же самые процедуры, которые мы уже создали, могут позаботиться обо всем этом.

Давайте начнем с создания новой переменной Base:

```
0 VALUE lbase
```

Мы будем использовать эту переменную вместо NumParams для вычисления смещения стека. Это подразумевает изменение двух ссылок на NumParams в LoadParam и StoreParam:

```
-- ------------------------------------------------------------
-- Load a parameter to the primary register
: loadparameter ( index -- )
        lbase SWAP - 2+ CELLS ( offset ) >R
        S" [ebp "  R> (.) $+  S"  +] -> eax  mov," $+ emitln ;
-- ------------------------------------------------------------
-- Store a parameter from the primary register  (call by value)
: storeparameter ( index -- )
        lbase SWAP - 2+ CELLS ( offset ) >R
        S" eax -> [ebp "  R> (.) $+  S"  +]  mov," $+ emitln ;
-- ------------------------------------------------------------
```

Идея состоит в том, что значение Base будет заморожено после того, как мы обработаем формальные параметры и не будет увеличиваться дальше когда новые, локальные,

переменные будут вставлены в таблицу идентификаторов. Об этом позаботится код в конце FormalList:

```
-- --------------------------------------------------------------
-- Process the formal parameter list of a procedure
: formallist ( -- )
        '(' match
        Look ')' <>
           IF   formalparam
                BEGIN  Look ',' =
                WHILE  ',' match formalparam
                REPEAT
        ENDIF
        ')' match fin
        #params TO lbase
        2 +TO #params ;
-- --------------------------------------------------------------
```

(Мы добавили четыре слова чтобы учесть адрес возврата и старый указатель кадра, который заканчивается между формальными параметрами и локальными переменными.)

Все что мы должны сделать дальше - это установить семантику объявления локальных переменных в синтаксическом анализаторе. Подпрограммы очень похожи на Decl и TopDecls:

```
-- --------------------------------------------------------------
-- Parse and translate a local data declaration
: locdecl ( -- ) 'v' match  getname addparam  fin ;
-- --------------------------------------------------------------
-- Parse and translate local declarations
: locdecls ( -- n )
        0
        BEGIN  Look 'v' =
        WHILE  locdecl  1+
        REPEAT ;
-- --------------------------------------------------------------
```

Заметьте, что LocDecls является функцией, возвращающей число локальных переменных в DoProc.

Затем мы изменим DoProc для использования этой информации:

```
-- --------------------------------------------------------------
-- Parse and translate a procedure declaration
: doProc ( -- )
        'p' match
        getname >R
        R@ 'p' addentry
        formallist
        locdecls ( #formals )
        R> SWAP ( name #formals ) procprolog
        beginblock
        procepilog
        clearparams ;
-- --------------------------------------------------------------
```

(Я сделал пару изменений, которые не были в действительности необходимы. Кроме небольшой реорганизации я переместил вызов Fin в FormalList а также в LocDecls. Не забудьте поместить его в конец FormalList.)

Обратите внимание на изменения при вызове ProcProlog. Новый параметр - это число слов (не байт) для распределения памяти. Вот новая версия ProcProlog:

```
-- --------------------------------------------------------------
-- Write the prolog for a procedure
: procprolog ( char #formals -- )
```

```
        >R S" SUBROUTINE " ROT CHAR-APPEND emitdln
        R> CELLS (.) S" 0  enter," $+ emitln ;
-- -----------------------------------------------------------
```

Сейчас должно работать. Добавьте эти изменения и посмотрите как они работают.(calls10.frt

ЗАКЛЮЧЕНИЕ

К этому моменту вы знаете как компилировать объявления и вызовы процедур с параметрами, передаваемыми по ссылке и по значению. Вы можете также обрабатывать локальные переменные. Как вы можете видеть, сложность состоит не в предоставлении механизма, а в определении какой механизм использовать. Стоит нам принять эти решения и код для трансляции в действительности не будет таким сложным.

Я не показал вам как работать с комбинацией локальных параметров и передачей параметров по ссылке, но это простое расширение того, что вы уже видели. Это просто немного более хлопотно и все, так как мы должны поддерживать оба механизма вместо одного. Я предпочел оставить это на потом, когда мы научимся работать с различными типами переменных.

Это будет следующая глава, которая появится ближе к Форуму. Увидимся.

14. Типы

ВВЕДЕНИЕ

В последней главе (Часть 13, Процедуры) я упомянул, что в ней и в следующей главе мы рассмотрим две возможности, которые помогут отделить игрушечный язык от настоящего, пригодного к использованию. В ней мы рассмотрели вызовы процедур. Многие из вас терпеливо ждали, начиная с Августа'89 когда я выдам вторую. Хорошо, вот она.

В этой главе мы поговорим о том, как работать с различными типами данных. Как и в последней главе, я не буду сейчас включать эти возможности непосредственно в компилятор TINY. Вместо этого я буду использовать тот же самый подход, который так хорошо служил нам в прошлом: использование только фрагментов синтаксического анализатора и односимвольных токенов. Как обычно, это позволит на добраться непосредственно до сути вопроса не продираясь сквозь массу ненужного кода. Так как основные проблемы при работе с множественными типами данных возникают в арифметических операциях, на них мы и сконцентрируем свое внимание.

Несколько предупреждений: Во-первых, есть некоторые типы, которые я не буду охватывать в этой главе. Здесь мы будем говорить только о простых, встроенных типах. Мы даже не будем работать с массивами, указателями или строками, я охвачу их в следующих нескольких главах.

Во-вторых, мы также не будем обсуждать и типы определяемые пользователем. Это будет значительно позже, по той простой причине, что я все еще не убедил себя, что эти определяемые пользователем типы должны быть в языке KISS. В более поздних главах я собираюсь охватить по крайней мере основные концепции определяемых пользователем типов, записей и т.д., просто для того, чтобы серия была полной. Но действительно ли они будут включены как часть KISS - все еще открытый вопрос. Я открыт для комментариев и предложений по этой теме.

Наконец, я должен предупредить вас: то, что мы собираемся сделать может добавить массу дополнительных сложностей и в синтаксический анализатор и в генерируемый код. Поддерживать различные типы достаточно просто. Сложность возникает когда вы добавляете правила преобразования между типами. Вообще-то, будет ли ваш компилятор простым или сложным зависит от способа, выбранного вами для определения правил преобразования

типов. Даже если вы решите запретить любые преобразования типов (как в Ada, например) проблема все еще остается, и она встроена в математику. Когда вы умножаете два коротких числа, к примеру, вы можете получить длинный результат.

Я подошел к этой проблеме очень осторожно, пытаясь сохранить простоту. Но мы не можем полностью избежать сложности. Как обычно случается, мы оказываемся перед необходимостью выбирить между качеством кода и сложностью и, как обычно, я предпочитаю выбрать самый простой подход.

ЧТО БУДЕТ ДАЛЬШЕ?

Прежде чем мы погрузимся в это занятие, я думаю вам бы хотелось знать что мы сейчас собираемся делать... особенно после того, как прошло столько много времени с прошлой главы.

Тем временем я не бездействовал. Я разбил компилятор на модули. Одна из проблем, с которыми я столкнулся, в том, что так как мы охватывали новые области и вследствие этого расширяли возможности компилятора TINY, он становился все больше и больше. Я понял пару глав назад, что это приводило к затруднениям и именно поэтому я возвратился к использованию только фрагментов компилятора в последней и этой главах. Кажется просто глупо заново воспроизводить код для, скажем, обработки булевых исключающих ИЛИ, когда тема дискуссии - передача параметров.

Очевидным способом получит свой пирог и съесть его также является разбиение компилятора на раздельно компилируемые модули и, конечно, модули Turbo Pascal являются для этого идеальным средством. Это позволит нам скрыть некоторый довольно сложный код (такой как полный синтаксический анализ арифметических и булевых выражений) в одиночный модуль и просто вытаскивать его всякий раз когда он необходим. При таком способе единственным кодом, который я должен буду воспроизводить в этих главах, будет код который непосредственно касается обсуждаемого вопроса.

Я также игрался с Turbo 5.5 который, конечно, включает Борландовские объектно-ориентированные расширения Паскаля. Я не решил, использовать ли эти возможности, по двум причинам. Прежде всего, многие из вас, кто следовал за этой серией, могут все еще не иметь 5.5 и я конечно не хочу вынуждать кого-либо пойти и купить новый компилятор только для того, чтобы завершить эту серию. Во-вторых, я не убежден, что ОО расширения имеют такое большое значение для этого приложения. Мы обсуждали кое-что из этого на форуме CLM на CompuServe, и пока что мы не нашли никакой убедительной причины для использования ОО конструкции. Это одна из тех областей, где я мог бы использовать некоторую обратную связь с читателями. Кто-нибудь хочет проголосовать за Turbo 5.5 и ООП?

В любом случае после следующих нескольких глав этой серии я планирую предоставить вам законченный набор модулей а также законченные функционирующие компиляторы. Планом фактически предусмотрено три компилятора: один для односимвольной версии TINY (для использования в наших экспериментах), один для TINY и один для KISS. Я достаточно четко выделил различия между TINY и KISS:

- TINY будет поддерживать только два типа данных: символьный и 16-разрядное целое число. Я могу также попробовать сделать что-нибудь со строками, так как без них компилятор был бы довольно бесполезным. KISS будет поддерживать все обычные простые типы, включая массивы и даже числа с плавающей точкой.
- TINY будет иметь только две управляющие конструкции IF и WHILE. KISS будет поддерживать очень богатый набор конструкций включая одну, которую мы не обсуждали здесь ранее... CASE.
- KISS будет поддерживать раздельно компилируемые модули.

Одно предостережение: так как я все еще не знаю достаточно об ассемблере для 80x86, все эти модули компилятора все еще будут написаны для поддержки кода 68000. Однако в

программах, которые я планирую вам представить, вся генерация кода была тщательно изолирована в отдельном модуле, так что любой предприимчивый студент смог бы перенастроить их на любой другой процессор. Эта задача "оставлена как упражнение для студента". Я сделаю предложение прямо здесь и сейчас: с человеком, который предоставит нам первый надежный перевод для 80x86, я буду счастлив обсудить коллективные авторские права и авторские отчисления от предстоящей книги.

Но хватит говорить. Давайте приступим к изучению типов. Как я сказал ранее, мы будем делать это как и в последней главе: выполняя эксперименты с использованием односимвольных токенов.

ТАБЛИЦА ИДЕНТИФИКАТОРОВ

Должно быть очевидным, что если мы собираемся работать с переменными различных типов, нам понадобится какое-то место для записи их типов. Очевидным средством для этого является таблица идентификаторов и мы уже использовали ее например для различия между локальными и глобальными переменными и между переменными и процедурами.

Структура таблицы идентификаторов для односимвольных токенов особенно проста и мы использовали ее прежде несколько раз. Для работы с ней, мы возьмем некоторые процедуры, которые мы использовали раньше.

Сначала, нам необходимо объявить саму таблицу идентификаторов:

```
-- Variable declarations -------------------------------------

0 VALUE Look                            -- lookahead character

CREATE ST         #26 CHARS ALLOT       -- symbol table
```

Затем мы должны удостовериться, что она инициализируется в процедуре Init:

```
-- -----------------------------------------------------------
: clearST ( -- ) ST #26 CHARS '? FILL ;
-- -----------------------------------------------------------
: init ( -- ) clearST  CR getchar skipwhite ;
-- -----------------------------------------------------------
```

Следующая процедура в действительности нам не нужна, но она будет полезна для отладки. Все, что она делает, это формирует дамп содержимого таблицы идентификаторов:

```
-- -----------------------------------------------------------
: ST[n]  ( n -- addr ) 'A' - ST + ;
: typeof ( char -- ) ST[n] C@ ;
-- Dump the symbol table
: .symbols ( -- )
        CR ." -- type --.--- name ---"
        'Z' 1+ 'A' ?DO  I ST[n] C@ IF CR  5 HTAB I typeof EMIT
                                       #16 HTAB I        EMIT
                                ENDIF
                  LOOP ;
-- -----------------------------------------------------------
```

В действительности не имеет значения, где вы поместите эту процедуру... я планирую группировать все подпрограммы таблицы идентификаторов вместе, так что я поместил ее сразу после процедур сообщений об ошибках.

Если вы осторожный тип (как я), вам возможно захотелось бы начать с тестовой программы, которая ничего не делает а просто инициализирует таблицу и затем создает ее дамп. Только для того, чтобы быть уверенным, что все мы находимся на одной волне, ниже я воспроизвожу всю программу, дополненную новыми процедурами. Заметьте, что эта версия включает поддержку пробелов:

```
-- -------------------------------------------------------------------------
0 VALUE Look                   -- lookahead character
CREATE ST        #26 CHARS ALLOT         -- symbol table
: clearST ( -- ) ST #26 CHARS '?' FILL ; -- clear symbol table
--
---------------------------------------------------------------------------
-----------------
-- Get type of symbol
: ST[n]  ( n -- addr ) 'A' - ST + ;
--
---------------------------------------------------------------------------
: typeof ( char -- )   ST[n] C@ ;
--
---------------------------------------------------------------------------
-- Dump symbol table
: .symbols ( -- )
        CR ." -- type --.--- name ---"
        'Z' 1+ 'A' ?DO  I ST[n] C@ IF CR  5 HTAB I typeof EMIT
                                        #16 HTAB I        EMIT
                                ENDIF
                  LOOP ;
--
---------------------------------------------------------------------------
: getchar  ( -- )          EKEY TO Look ;                        -- read new
character from input stream
: error    ( c-addr u -- ) CR ^G EMIT ." Error: " TYPE ." ." ;  -- report an error
: aborts   ( c-addr u -- ) error ABORT ;                         -- report error and
halt
: expected ( c-addr u -- ) S"  expected" $+ aborts ;            -- report what was
expected
: alpha?   ( char -- tf )  >UPC 'A' 'Z' 1+ WITHIN ;             -- recognize an
alpha character
: digit?   ( char -- tf )  '0' '9' 1+ WITHIN ;                  -- recognize a
decimal digit
: alnum?   ( char -- tf )  DUP alpha? SWAP digit? OR ;          -- recognize
alphanumeric
: white?   ( char -- tf )  DUP  BL =  SWAP Tab =  OR ;          -- Recognize white
space
: addop?   ( char -- tf )  DUP '+' =  SWAP '-' =  OR ;          -- test for AddOp
: mulop?   ( char -- tf )  DUP '*' =  SWAP '/' =  OR ;          -- test for MulOp
: orop?    ( char -- tf )  DUP '|' =  SWAP '~' =  OR ;          -- recognize an OR
operand
--
---------------------------------------------------------------------------
-- Recognize a relop
: relop? ( char -- tf )
        DUP  '=' =
        OVER '#' = OR
        OVER '<' = OR
        SWAP '>' = OR ;
--
---------------------------------------------------------------------------
-- Skip white space
: skipwhite ( -- )  BEGIN  Look white?  WHILE  getchar REPEAT ;
-- --------------------------------------------------------------
-- Skip over EOL
: fin ( -- ) Look ^M = IF  getchar  Look ^J = IF  getchar  ENDIF ENDIF ;
--
---------------------------------------------------------------------------
-- Match a specific input character
: match ( char -- )
        DUP Look = IF  DROP getchar skipwhite
                 ELSE  S" `" ROT CHAR-APPEND &' CHAR-APPEND expected
                ENDIF ;
--
---------------------------------------------------------------------------
-- Get an identifier
```

```
: getname ( -- char )
        Look alpha? 0= IF  S" Identifier" expected  ENDIF
        Look >UPC  getchar skipwhite ;
--
------------------------------------------------------------------------------------
-- Get a number
: getnum ( -- num )
        Look digit? 0= IF  S" Integer" expected  ENDIF
        Look  getchar skipwhite ;
--
------------------------------------------------------------------------------------
-- Initialize everything
: init   ( -- ) clearST CR getchar skipwhite ;
--
----------------------------------------------------------------------------------
-- Main program
: types1 ( -- )  init .symbols ;
-- ---------------------------------------------------------------------------------
```

ОК, запустите эту программу(types1.frt. Вы должны получить (очень быстро) распечатку всех букв алфавита (потенциальных идентификаторов) сопровождаемых вопросительным знаком. Не очень захватывающе, но это только начало.

Конечно, вообще-то мы хотим видеть типы только тех переменных, которые были определены. Мы можем устранить другие добавив в DumpTable условие IF. Измените цикл следующим образом:

```
    I typeof '?' <> IF CR  5 HTAB I typeof EMIT
                         #16 HTAB I        EMIT
                 ENDIF
```

Теперь запустите программу снова. Что вы получили?

Хорошо, это даже более скучно чем раньше! Сейчас вообще ничего не выводится, так как в данный момент ни одно из имен не было обьявлено. Мы можем немного приправить результат вставив в основную программу несколько операторов, объявляющих несколько записей. Попробуйте такие:

```
        'a' 'A' ST[n] C!
        'b' 'P' ST[n] C!
        'c' 'X' ST[n] C!
```

На этот раз, когда вы запустите программу, вы должны получить распечатку, показывающую, что таблица идентификаторов работает правильно.

ДОБАВЛЕНИЕ ЗАПИСЕЙ

Конечно, заполнение таблицы напрямую - довольно плохая практика и она не сможет хорошо нам послужить в будущем. То, что нам нужно, это процедура, добавляющая записи в таблицу. В то же самое время мы знаем, что нам будет необходимо тестировать таблицу для проверки, что мы не объявляем повторно переменную, которая уже используется (что легко может случиться при наличии всего 26 вариантов!). Для поддержки всего это введите следующие новые процедуры:

```
-- ------------------------------------------------------------------
-- Report if a variable is in the table
: intable ( char -- )   typeof '?' <> ;
-- ------------------------------------------------------------------
-- Check for a duplicate variable name
: name.aborts ( c-addr1 u1 char )
        >R S"  `" $+
        R> CHAR-APPEND &' CHAR-APPEND aborts ;
-- ------------------------------------------------------------------
```

```
: .duplicate ( char -- ) S" Duplicate identifier" ROT name.aborts ;
: checkdup   ( char -- ) DUP intable 0= IF DROP EXIT ENDIF .duplicate ;
-- ---------------------------------------------------------------------
-- Add a new entry to symbol table
: addentry ( char type -- )  OVER checkdup  SWAP ST[n] C! ;
-- ---------------------------------------------------------------------
```

Теперь измените три строки в основной программе следующим образом:

```
        'a' 'A' addentry
        'b' 'P' addentry
        'c' 'X' addentry
```

и запустите программу снова.(types4.frt) Работает? Тогда у нас есть подпрограммы таблицы идентификаторов, необходимые для поддержки нашей работы с типами. В следующем разделе мы начнем их использовать на практике.

РАСПРЕДЕЛЕНИЕ ПАМЯТИ

В других программах, подобных этой, включая сам компилятор TINY, мы уже обращались к вопросу объявления глобальных переменных и кода, генерируемого для них. Давайте создадим здесь урезанную версию "компилятора", чья единственная функция - позволить нам объявлять переменные. Помните, синтаксис для объявления:

```
     <data decl> ::= VAR <identifier>
```

Снова, мы можем вытащить массу кода из предыдущих программ. Следующий код - это урезанные версии тех процедур. Они значительно упрощены, так как я удалил такие тонкости как списки переменных и инициализаторы. Обратите внимание, что в процедуре Alloc новый вызов AddEntry будет также заботиться о проверке двойных объявлений:

```
-- ---------------------------------------------------------------
-- Allocate storage for a variable
: alloc ( char -- )
        'v' OVER addentry
        S" CREATE " ROT CHAR-APPEND S"  0 ," $+ emitdln ;
-- ---------------------------------------------------------------
-- Parse and translate a data declaration
: decl ( -- ) 'v' match getname alloc ;
-- ---------------------------------------------------------------
-- Parse and translate global declarations
: topdecls ( -- )
        BEGIN   Look '.' <>
        WHILE   CASE Look
                 'v' OF decl ENDOF
                        S" Unrecognized keyword "
                        Look CHAR-APPEND aborts
                ENDCASE
                fin
        REPEAT ;
-- ---------------------------------------------------------------
```

Теперь, в основной программе добавьте вызов TopDecl и запустите программу.(types5.frt) Попробуйте распределить несколько переменных и обратите внимание на полученный сгенерированный код. Для вас это пройденный этап, поэтому результат должен выглядеть знакомым. Заметьте из кода для TopDecls что программа завершается точкой.

Пока вы здесь, попробуйте объявить две переменные с одинаковыми именами и проверьте что синтаксический анализатор отлавливает ошибку.

ОБЪЯВЛЕНИЕ ТИПОВ

Распределение памяти различных размеров не сложнее чем изменение процедуры TopDecl для распознавания более чем одного ключевого слова. Здесь необходимо принять ряд решений, с точки зрения того, каков должен быть синтаксис и т.п., но сейчас я собираюсь отложить все эти вопросы и просто объявить не подлежащий утверждению указ что наш синтаксис будет таким:

```
<data decl> ::= <typename>  <identifier>
```

где:

```
<typename> ::= BYTE | WORD | LONG
```

(По удивительному совпадению, первые буквы этих наименований оказались те же самыми что и спецификации длины ассемблерного кода 68000, так что такой выбор съэкономит нам немного работы.)

Мы можем создать код, который позаботится об этих объявлениях, внеся всего лишь небольше изменения. Обратите внимание, что в подпрограммах, показанных ниже, я отделил генерацию код в Alloc от логической части. Это соответствует нашему желанию изолировать машино-зависимую часть компилятора.

```
-- ------------------------------------------------------------------------
-- Generate code for allocation of a variable
: allocvar ( char -- ) S" CREATE " ROT CHAR-APPEND S"  0 ," $+ emitdln ;
-- ------------------------------------------------------------------------
-- Allocate storage for a variable
: alloc ( N type -- ) OVER addentry  allocvar ;
-- ---------------------------------------------------------------
-- Parse and Translate a Data Declaration
: decl ( -- )  getname ( type) getname ( N) SWAP alloc ;
-- ---------------------------------------------------------------
-- Parse and translate global declarations
: topdecls ( -- )
        BEGIN   Look '.' <>
        WHILE   CASE Look
                 'b' OF decl ENDOF
                 'w' OF decl ENDOF
                 'l' OF decl ENDOF
                        S" Unrecognized keyword "
                        Look CHAR-APPEND aborts
                ENDCASE
                fin
        REPEAT ;
-- ---------------------------------------------------------------
```

Внесите показанные изменения в эти процедуры и испытайте программу.(types6.frt) Используйте одиночные символы "b", "w" и "l" как ключевые слова (сейчас они должны быть в нижнем регистре). Вы увидите, что в каждом случае мы выделяем память соответствующего объема. Обратите внимание, глядя на дамп таблицы идентификаторов, что размеры также сохранены для использования позже. Какого использования? Хорошо, это тема остальной части этой главы.

ПРИСВАИВАНИЯ

Теперь, когда мы можем объявлять переменные различных размеров, очевидно что мы должны иметь возможность что-то с ними делать. На первый раз, давайте просто попробуем загружать их в наш рабочий регистр D0. Имеет смысл использовать ту же самую идею, которую мы использовали для Alloc, т.е. сделаем процедуру загрузки, которая может загружать переменные нескольких размеров. Нам также необходимо продолжать изолировать машино-зависимое содержимое. Процедура загрузки выглядит так:

```
-- --------------------------------------------------------------
-- Load primary register from variable
: loadvariable  ( char type -- )
        >R
        DUP intable 0= IF  .undefined  ENDIF
        CASE R>
          'b' OF S"  byte-ptr -> al  mov,"   ENDOF
          'w' OF S"  word-ptr -> ax  mov,"   ENDOF
          'l' OF S"  dword-ptr -> eax  mov," ENDOF
        ENDCASE
        ROT CHAR-PREPEND emitln ;
-- --------------------------------------------------------------
```

По крайней мере для 68000, многие команды оказываются командами MOVE. Было бы полезно создать отдельный генератор кода только для этих инструкций и затем вызывать его когда необходимо:

```
-- --------------------------------------------------------------
-- Recognize a legal variable type
: vartype ( char -- tf ) DUP 'B' = OVER 'W' = OR SWAP 'L' = OR ;
-- --------------------------------------------------------------
```

Обратите внимание, что эти две подпрограммы - строго генераторы кода; они не имеют проверки ошибок и другой логики. Чтобы завершить картинку, нам необходим еще один программный уровень, который предоставляет эти функции.

Прежде всего, мы должны удостовериться, что типы, с которыми мы работаем - загружаемого типа. Это звучит как работа для другого распознавателя:

```
-- --------------------------------------------------------------
-- Recognize a legal variable type
: vartype ( char -- tf ) DUP 'B' = OVER 'W' = OR SWAP 'L' = OR ;
-- --------------------------------------------------------------
```

Затем, было бы хорошо иметь подпрограмму, которая извлечет тип переменной из таблицы идентификаторов в то же время проверяя его на допустимость:

```
-- --------------------------------------------------------------
-- Get a variable type from the symbol table
: vartype ( ch -- char )
        DUP typeof LOCALS| Typ name |
        Typ vartype? 0= IF  S" Identifier `" name CHAR-APPEND
                            S" ' is not a variable" $+ aborts
                     ENDIF
        Typ ;
-- --------------------------------------------------------------
```

Вооруженная этими инструментами, процедура, выполняющая загрузку переменной, становится тривиальной:

```
-- --------------------------------------------------------------
-- Load a variable to the primary register
: Load ( name -- ) DUP vartype loadvariable ;
-- --------------------------------------------------------------
```

(Примечание для обеспокоившихся: я знаю, знаю, все это очень неэффективно. В промышленной программы мы, возможно, предприняли бы шаги чтобы избежать такого глубокого вложения вызовов процедур. Не волнуйтесь об этом. Это упражнение, помните? Более важно сделать его правильно и понять его, чем получить неправильный ответ но быстро. Если вы закончите свой компилятор и обнаружите, что вы несчастны от его быстродействия, вы вольны вернуться и доработать код для более быстрой работы).

Было бы хорошей идеей протестировать программу сейча(types7.frt).c. Так как мы пока не имеем процедуры для работы с операциями присваивания, я просто добавил строки:

```
        'A' load
        'B' load
        'C' load
        'X' load
```

в основную программу. Таким образом, после того, как раздел объявления завершен, они будут выполнены чтобы генерировать код для загрузки. Вы можете поиграть с различными комбинациями объявлений чтобы посмотреть как обрабатываются ошибки.

Я уверен, что вы не будете удивлены, узнав, что сохранение переменных во многом подобно их загрузке. Необходимые процедуры показаны дальше:

```
-- ---------------------------------------------------------------
-- Store primary to variable
: storevariable ( char type -- )
        >R
        DUP checktable
        CASE R>
          'B' OF S" al -> "  ROT CHAR-APPEND S"  byte"  ENDOF
          'W' OF S" ax -> "  ROT CHAR-APPEND S"  word"  ENDOF
          'L' OF S" eax -> " ROT CHAR-APPEND S"  dword" ENDOF
        ENDCASE
        S" -ptr mov," $+ $+ emitln ;
-- --------------------------------------------------------------
-- Store a variable from the primary register
: store ( name -- ) DUP vartype storevariable ;
-- --------------------------------------------------------------
```

Вы можете проверить их таким же образом, что и загрузку.(types8.frt)

Теперь, конечно, достаточно легко использовать их для обработки операций присваивания. Что мы сделаем - создадим специальную версию процедуры Block, которая поддерживает только операции приваивания, а также специальную версию Expression, которая поддерживает в качестве допустимых выражений только одиночные переменные. Вот они:

```
-- --------------------------------------------------------------
-- Parse and translate an expression
: expression ( -- ) getname load ;
-- --------------------------------------------------------------
-- Parse and translate an assignment statement
: assignment ( -- )
        getname LOCAL name
        '=' match
        expression
        name store ;
-- --------------------------------------------------------------------
-- Parse and translate a block of statements
: block ( -- ) BEGIN  Look '.' <>  WHILE  assignment fin  REPEAT ;
-- --------------------------------------------------------------------
```

(Стоит заметить, что новые процедуры, которые позволяют нам манипулировать типами, даже проще и яснее чем те, что мы видели ранее. Это в основном блягодаря нашим усилиям по изоляции подпрограмм генерации кода.)

Есть одна небольшая назойливая проблема. Прежде мы использовали завершающую точку Паскаля чтобы выбраться из процедуры TopDecl. Теперь это неправильный символ... он использован для завершения Block. В предудущих программах мы использовали для выхода символ BEGIN (сокращенно "b"). Но он теперь используется как символ типа.

Решение, хотя и является отчасти клуджем, достаточно простое. Для обозначения BEGIN мы будем использовать 'B' в верхнем регистре. Так что измените символ в цикле WHILE

внутри TopDecl с "." на "B" и все будет прекрасно.

Теперь мы можем завершить задачу, изменив основную программу следующим образом:

```
-- -------------------------------------------------------------
-- Main program
: types9 ( -- )
        init
        topdecls
        'B' match fin
        block
        .symbols ;
-- -------------------------------------------------------------
```

(Обратите внимание, что я должен был расставить несколько обращений к Fin чтобы избежать проблем переносов строк.)

OK, запустите эту программу.(types9.frt) Попробуйте ввести:

```
     ba         { byte a }   *** DON'T TYPE THE COMMENTS!!! ***
     wb         { word b }
     lc         { long c }
     B          { begin  }
     a=a
     a=b
     a=c
     b=a
     b=b
     b=c
     c=a
     c=b
     c=c
     .
```

Для каждого объявления вы должны получить сгенерированный код, распределяющий память. Для каждого присваивания вы должны получить код который загружает переменную корректного размера и сохраняет ее, также корректного размера.

Есть только одна небольшая проблема: сгенерированный код неправильный!

Взгляните на код для a=c:

```
        C dword-ptr -> eax  mov,
        al -> A byte-ptr mov,
```

Этот код корректный. Он приведет к сохранению младших восьми бит C в A, что является примлемым поведением. Это почти все, что мы можем ожидать.

Но теперь, взгляните на противоположный случай. Для c=a генерируется такой код:

```
        A byte-ptr -> al  mov,
        eax -> C dword-ptr mov,
```

Это не правильно. Он приведет к сохранению байтовой переменной A в младших восьми битах D0. Согласно правилам для процессора 68000 старшие 24 бита останутся неизменными. Это означаем, что когда мы сохраняем все 32 бита в C, любой мусор, который был в этих старших разрядах, также будет сохранен. Нехорошо.

То, с чем мы сейчас столкнулись назвается проблемой преобразования типов или приведением.

Прежде, чем мы сделаем что-либо с переменными различных типов, даже если это просто их копирование, мы должны быть готовы встретиться с этой проблемой.   Это не самая простая часть компилятора. Большинство ошибок, которые я видел в промышленных компиляторах, имели отношение к ошибкам преобразования типов для некоторой неизвестной комбинации аргументов. Как обычно, существует компромисс между

сложностью компилятора и потенциальным качеством сгенерированного кода, и, как обычно, мы выберем путь, который сохранит компилятор простым. Я думаю вы надете, что с таким подходом мы можем удерживать потенциальную сложность под достаточным контролем.

ТРУСЛИВЫЙ ВЫХОД

Прежде, чем мы заберемся в детали (и потенциальную сложность) преобразования типов, я хотел бы, чтобы вы видели, что существует один суперпростой способ решения проблемы: просто переводить каждую переменную в длинное целое во время загрузки!

Для этого достаточно добавить всего одну строку в LoadVar, хотя, если мы не собираемся полностью игнорировать эффективность, она должна ограничиваться проверкой IF. Вот измененная версия:

```
-- ----------------------------------------------------------------
-- Load a variable to primary register
: loadvariable  ( char type -- )
        >R
        DUP checktable
        CASE R>
          'B' OF S"  byte-ptr -> eax  movzx," ENDOF
          'W' OF S"  word-ptr -> eax  movzx," ENDOF
          'L' OF S"  dword-ptr -> eax  mov,"  ENDOF
        ENDCASE
        ROT CHAR-PREPEND emitln ;
-- ----------------------------------------------------------------
```

(Обратите внимание, что StoreVar не нуждается в подобном изменении).

Если вы выполните некоторые тесты с этой новой версией, вы обнаружите, что теперь все работает правильно, хотя иногда неэффективно. К примеру, рассмотрим случай a=b (для тех же самых объявлений, что показаны выше). Теперь сгенерированный код становится:

```
CLR.L D0
MOVE.W B(PC),D0
LEA  A(PC),A0
MOVE.B D0,(A0)
```

В этом случае CLR оказывается ненужной, так как результат помещается в байтовую переменную. Небольшая доработка помогла бы нам улучшить его. Однако, все это не так уж плохо, и это типичного рода неэффективность, которую мы видели прежде в нехитрых компиляторах.

Я должен подчеркнуть, что устанавливая старшие разряды в нуль, мы фактически обрабатываем числа как целые числа без знака. Если вместо этого мы хотим обрабатывать их как целые числа со знаком (более вероятный случай) мы должны делать расширение знака после загрузки. Просто для того, чтобы обернуть эту часть дискуссии милой красной ленточкой, давайте изменим LoadVar как показано ниже:

```
-- ----------------------------------------------------------------
-- Load a Variable to Primary Register }
: loadvariable  ( char type -- )
        >R
        DUP checktable
        CASE R>
          'B' OF S"  byte-ptr -> eax  movzx," ENDOF
          'W' OF S"  word-ptr -> eax  movsx," ENDOF
          'L' OF S"  dword-ptr -> eax  mov,"  ENDOF
        ENDCASE
        ROT CHAR-PREPEND emitln ;
-- ----------------------------------------------------------------
```

В этой версии байт обрабатывается как беззнаковое число (как в Паскале и Си) в то время как слово обрабатывается как знаковое(types10.frt).

БОЛЕЕ ПРИЕМЛЕМОЕ РЕШЕНИЕ

Как мы видели, перевод каждой переменной в длинное слово пока она находится в памяти решает проблему, но это едва ли может быть названо эффективным и, возможно, не было бы приемлемым даже для тех из нас, кто требует не обращать внимания на эффективность. Это означает, что все арифметические операции будут выполняться с 32-битной точностью, что удвоит время выполнения для большинства операций и сделает его еще больше для умножения и деления. Для этих операций мы должны были бы вызывать подпрограммы, даже если данные были бы байтом или словом. Все это слишком походит на уловку, так как уводит нас от всех настоящих проблем.

ОК, значит это решение плохое. Есть ли еще относительно простой способ получить преобразование данных? Можем ли мы все еще сохранять простоту?

Да, действительно. Все, что нам нужно сделать - выполнить преобразование с другого конца... т.е. мы выполняем преобразование на выходе, когда данные сохраняются, а не на входе.

Но запомните, часть присваивания, отвечающая за хранение, в значительной степени независима от загрузки данных, о которой заботится процедура Expression. Вообще, выражение может быть произвольно сложным, поэтому как может процедура Assignment знать, какой тип данных оставлен в регистре D0?

Снова, ответ прост: Мы просто спросим об этом процедуру Expression! Ответ может быть возвращен как значение функции.

Все это требует изменения некоторых процедур, но эти изменения, как и сам метод, совсем простые. Прежде всего, так как мы не требуем чтобы LoadVar выполнял всю работу по преобразованию, давайте возвратимся к простой версии:

```
-- --------------------------------------------------------------
-- Load a variable to primary register
: loadvariable  ( char type -- )
        >R
        DUP checktable
        CASE R>
          'B' OF S"  byte-ptr -> al  mov,"    ENDOF
          'W' OF S"  word-ptr -> ax  mov,"    ENDOF
          'L' OF S"  dword-ptr -> eax  mov,"  ENDOF
        ENDCASE
        ROT CHAR-PREPEND emitln ;
-- --------------------------------------------------------------
```

Затем, давайте добавим новую процедуру, которая будет выполнять преобразование из одного типа в другой:

```
-- --------------------------------------------------------------
-- Convert a data item from one type to another
: convert ( source dest -- )
        2DUP = IF  2DROP EXIT  ENDIF
        SWAP 'B' = IF  S" al -> eax  movzx," emitln  ENDIF
             'L' = IF  S" ax -> eax  movsx," emitln  ENDIF ;
-- --------------------------------------------------------------
```

Затем, мы должны реализовать логику, требуемую для загрузки и сохранения переменной любого типа. Вот подпрограммы для этого:

```
-- --------------------------------------------------------------
-- Load a variable to the primary register
: load ( name -- typ ) DUP vartype DUP >R loadvariable R> ;
```

```
-- -------------------------------------------------------------
-- Store a variable with type T1 from the primary register
: store ( name T1 -- )
        OVER vartype DUP >R convert R>
        ( name T2 -- ) storevariable ;
-- -------------------------------------------------------------
```

Обратите внимание, что Load является функцией, которая не только выдает код для загрузки, но также возвращает тип переменной. Таким образом, мы всегда знаем, с каким типом данных мы работаем. Когда мы выполняем Store, мы передаем ей текущий тип переменной в D0. Так как Store также знает тип переменной назначения, она может выполнить преобразование необходимым образом.

Вооруженная всеми этими новыми подпрограммами, реализация нашего элементарного присваивания по существу тривиальна. Процедура Expression теперь становится функцией возвращающей тип выражения в процедуру Assignment:

```
-- -------------------------------------------------------------
-- Parse and translate an expression
: expression ( -- type ) getname load ;
-- -------------------------------------------------------------
-- Parse and translate an assignment statement
: assignment ( -- )
        getname LOCAL name
        '=' match
        name expression store ;
-- -------------------------------------------------------------
```

Снова, заметьте как невероятно просты эти две подпрограммы. Мы изолировали всю логику типа в Load и Store и хитрость с передачей типа делает остальную работу чрезвычайно простой. Конечно, все это для нашего специального, тривиального случая с Expression. Естественно, для общего случая это будет более сложно. Но теперь вы смотрите на финальную версию процедуры Assignment!

Все это выглядит как очень простое и ясное решение, и действительно это так. Откомпилируйте эту программу и выполните те же самые тесты, что и ранее(types11.frt. Вы увидите, что все типы данных преобразованы правильно и здесь немного, если вообще есть, зря потраченных инструкций. Только преобразование "байт-длинное слово" использует две инструкции когда можно было бы использовать одну, и мы могли бы легко изменить Convert для обработки этого случая.

Хотя мы в этом случае не рассматривали переменные без знака, я думаю вы можете видеть, что мы могли бы легко исправить процедуру Convert для работы и с этими типами. Это "оставлено как упражнение для студента".

ЛИТЕРАЛЬНЫЕ АРГУМЕНТЫ

Зоркие читатели могли бы отметить, однако, что мы еще даже не имеем правильной формы простого показателя, потому что мы не разрешаем загрузку литеральных констант, только переменных. Давайте исправим это сейчас.

Для начала нам понадобится функция GetNum. Мы уже видели ее несколько версий, некоторые возвращают только одиночный символ, некоторые строку, а некоторые целое число. Та, которая нам здесь нужна будет возвращать длинное целое, так что она может обрабатывать все, что мы ей подбросим. Обратите внимание, что здесь не возвращается никакой информации о типах: GetNum не интересуется тем, как будет использоваться число:

```
-- -------------------------------------------------------------
-- Get a number
: getnum ( -- num )
        Look digit? 0= IF  S" Integer" expected  ENDIF
        0 >R
```

```
        BEGIN  Look digit?
        WHILE  R> #10 *  Look '0' - + >R
               getchar
        REPEAT R> skipwhite ;
-- ----------------------------------------------------------------
```

Теперь, когда работаем с литералами, мы имеем одну небольшую проблему. С переменными мы знаем какого типа они должны быть потому что они были объявлены с таким типом. Мы не имеем такой информации о типе для литералов. Когда программист говорит "-1", означает ли это байт, слово или длинное слово? Мы не имеем никаких сведений. Очевидным способом было бы использование наибольшего возможного типа, т.е. длинного слова. Но это плохая идея, потому что когда мы примемся за более сложные выражения, мы обнаружим, что это заставит каждое выражение включающее литералы, также переводить в длинное.

Лучшим подходом было бы выбрать тип, основанный на значении литерала, как показано далее:

```
-- ---------------------------------------------------------------
-- Load a constant to the primary register
: loadnumber ( n -- tf )
        0 LOCALS| typ N |
        N ABS #127 <= IF  'B' TO typ
                    ELSE  N ABS #32767 <= IF  'W'
                                        ELSE  'L'
                                       ENDIF  TO typ
                   ENDIF
        N typ loadconstant
        typ ;
-- ----------------------------------------------------------------
```

(Я знаю, знаю, база числа не является в действительности симметричной. Вы можете хранить -128 в одиночном байте и -32768 в слове. Но это легко исправить и не стоит затраченного времени или дополнительной сложности возиться с этим сейчас. Стоящая мысль.)

Заметьте, что LoadNum вызывает новую версию подпрограммы генерации кода LoadConst, которая имеет дополнительный параметр для определения типа:

```
-- ---------------------------------------------------------------
-- Load a constant to the primary register
: loadconstant ( N Typ -- )
        >R (.)
        CASE R>
          'B' OF  S"  b# -> al  mov,"  ENDOF
          'W' OF  S"  d# -> ax  mov,"  ENDOF
          'L' OF  S"  d# -> eax  mov," ENDOF
        ENDCASE  $+ emitln ;
-- ---------------------------------------------------------------
```

Теперь мы можем изменить процедуру Expression для использования двух возможных видов показателей:

```
-- ---------------------------------------------------------------
-- Parse and translate an expression
: expression ( -- type )
        Look alpha? IF  getname load
                  ELSE  getnum loadnumber
                 ENDIF ;
-- ---------------------------------------------------------------
```

(Вау, это, уверен, не причинило слишком большого вреда! Всего несколько дополнительных строк делают всю работу.)

ОК, соберите этот код в вашу программу и испытайте ее.(types12.frt) Вы увидите, что она теперь работает и для переменных и для констант как допустимых выражений.

АДДИТИВНЫЕ ВЫРАЖЕНИЯ

Если вы следовали за этой серией с самого начала, я уверен вы знаете, что будет дальше. Мы расширим форму выражения для поддержки сначала аддитивных выражений, затем мультипликативных, а затем общих выражений со скобками.

Хорошо, что мы уже имеем модель для работы с этими более сложными выражениями. Все, что мы должны сделать, это удостовериться, что все процедуры, вызываемые Expression, (Term, Factor и т.д.) всегда возвращают идентификатор типа. Если мы сделаем это, то структура программы едва ли вообще изменится.

Первый шаг прост: мы должны переименовать нашу существующую версию Expression в Term, как мы делали много раз раньше и создать новую версию Expression:

```
-- --------------------------------------------------------------
-- Parse and translate an expression
: expression ( -- typ )
        Look addop? IF unop ELSE term ENDIF LOCAL typ
        BEGIN   Look addop?
        WHILE   typ _push
                CASE Look
                 '+' OF  typ add      TO typ  ENDOF
                 '-' OF  typ subtract TO typ  ENDOF
                ENDCASE
        REPEAT  typ ;
-- --------------------------------------------------------------
```

Обратите внимание, как в этой подпрограмме каждый вызов процедуры стал вызовом функции и как локальная переменная Typ модифицируется при каждом проходе.

Обратите внимание также на новый вызов функции Unop, которая позволяет нам работать с ведущим унарным минусом. Это изменение не является необходимым... мы все еще можем использовать форму более похожую на ту, что мы использовали ранее. Я решил представить Unop как отдельную подпрограмму потому что позднее это позволит производить несколько лучший код, чем мы делали. Другими словами, я смотрю вперед на проблему оптимизации.

Для этой версии, тем не менее, мы сохраним тот же самый примитивный старый код, который делает новую подпрограмму тривиальной:

```
-- ---------------------------------------------------------------
-- Process a term with leading unary operator
: unop ( -- typ )
        _clear
        'W' ;
-- ---------------------------------------------------------------
```

Процедура Push - это подпрограмма генерации кода, которая теперь имеет параметр, указывающий тип.

```
-- ---------------------------------------------------------------
-- Push primary onto stack
: _push ( size -- )
        CASE
          'B' OF  S"  al  push,"  ENDOF
          'W' OF  S"  ax  push,"  ENDOF
          'L' OF  S"  eax  push," ENDOF
        ENDCASE  emitln ;
-- ---------------------------------------------------------------
```

Теперь давайте взглянем на функции Add и Subtract. В более старых версиях этих подпрограмм мы позволяем им вызывать подпрограммы генерации кода PopAdd и PopSub. Мы продолжим делать это, что делает сами функции чрезвачайно простыми:

```
-- ---------------------------------------------------------

-- Recognize and translate an add
: add ( typ1 -- typ2 )
        '+' match
        term _popadd ;
-- ---------------------------------------------------------------
-- Recognize and translate a subtract
: subtract ( typ1 -- typ2 )
        '-' match
        term _popsub ;
-- ---------------------------------------------------------------
```

Но простота обманчива, поскольку мы переложили всю логику на PopAdd и PopSub, которые больше не являются просто подпрограммами генерации кода. Они также должны теперь заботиться о необходимых преобразованиях типов.

Какие это преобразования? Простые: оба аргумента должны иметь тот же самый размер и результат также такой размер. Меньший из двух параметров должен быть "приведен" до размера большего.

Но это представляет небольшую проблему. Если переводимый параметр - второй (т.е. в основном регистре D0) мы в отличной форме. Если же нет, мы в затруднении: мы не можем изменить размер данных, которые уже затолкнуты в стек.

Решение простое, но немного болезненное: мы должны отказаться от этих красивых инструкций "вытолкнуть данные и что-нибудь с ними сделать", заботливо предоставленных Motorola.

Альтернативой является назначение вторичного регистра, в качестве которого я выбрал R7. (Почему не R1? Потому, что для других регистров у меня есть планы на будущее.)

Первый шаг в этой новой структуре - представить процедуру Pop, аналогичную Push. Эта процедура будет всегда выталкивать верхний элемент стека в D7:

```
-- ---------------------------------------------------------------
-- Pop stack into secondary register
: _pop ( size -- )
        CASE
          'B' OF  S"  bl  pop,"  ENDOF
          'W' OF  S"  bx  pop,"  ENDOF
          'L' OF  S"  ebx  pop," ENDOF
        ENDCASE  emitln ;
-- ---------------------------------------------------------------
```

Общая идея состоит в том, что все "Pop-Op" подпрограммы могут вызывать ее. Когда это сделано, мы будем иметь оба операнда в регистрах, поэтому мы можем перевести любой нужный нам. Для работы процедуре Convert необходим другой аргумент, имя регистра:

```
-- ---------------------------------------------------------------
-- Convert a data item from one type to another
: convert ( source dest reg$ -- )
        DLOCAL regname
        LOCALS| dest src |
        src dest = ?EXIT
        regname S" eax" COMPARE 0=
            IF  dest 'L' = src 'B' = AND
                IF  S" al -> eax  movzx," emitln  EXIT  ENDIF
                src  'B' = IF  S" al -> eax  movzx,"  emitln  ENDIF
                dest 'L' = IF  S" ax -> eax  movsx,"  emitln  ENDIF
```

```
                EXIT
         ENDIF
        regname S" edx" COMPARE 0=
            IF  dest 'L' = src 'B' = AND
                IF  S" dl -> edx  movzx," emitln  EXIT  ENDIF
                src  'B' = IF  S" dl -> edx  movzx,"  emitln  ENDIF
                dest 'L' = IF  S" dx -> edx  movsx,"  emitln  ENDIF
                EXIT
         ENDIF
         ( .. etcetera ) ;
-- ----------------------------------------------------------------
```

Следующая функция выполняет пребразование, но только если текущий тип T1 меньше по размеру, чем желаемый тип T2. Это функция, возвращающая конечный тип, позволяющий нам знать, что она решила:

```
-- ----------------------------------------------------------------
-- Promote the size of a register value
: promote ( T1 T2 reg$ -- tf )
        2OVER = IF  3DROP EXIT  ENDIF
        2SWAP LOCALS| T2 T1 |
        T1 'B' =
        T1 'W' = T2 'L' = AND
        OR IF  T1 T2 2SWAP convert  T2
         ELSE  2DROP T1
        ENDIF ;
-- ----------------------------------------------------------------
```

Наконец, следующая функция приводит два регистра к одному типу:

```
-- ----------------------------------------------------------------
-- Force both arguments to same type
: sametype ( T1 T2 -- typ )
        2DUP S" edx" promote ( T1 T2 typ)
        SWAP S" eax" promote NIP ;
-- ----------------------------------------------------------------
```

Эти новые подпрограммы дают нам заряд, необходимы нам чтобы разложить PopAdd и PopSub:

```
-- ----------------------------------------------------------------
-- Generate code to add primary to the stack
: _popadd ( T1 T2 -- typ )
        OVER _pop
        sametype ( typ)
        DUP _genadd _popadd ;
-- ----------------------------------------------------------------
-- Generate code to subtract primary from the stack
: _popsub ( T1 T2 -- type )
        OVER _pop
        sametype ( typ)
        DUP _gensub _popsub ;
-- ----------------------------------------------------------------
```

После всех этих приготовлений, в конечном результате нет почти ничего кульминационного. Снова, вы можете видеть что логика совершенно проста. Все что делают эти две подпрограммы - выталкивают вершину стека в D7, приводят два операнда к одному размеру и затем генерируют код.

Обратите внимание на две новые подпрограммы генерации кода GenAdd и GenSub. Они являются остаточной формой оригинальных PopAdd и PopSub. Т.е. они являются чистыми генераторами кода, производящими сложение и вычитание регистров:

```
-- ----------------------------------------------------------------
```

```
-- Add top of stack to primary
: _genadd ( size -- )
        CASE
          'B' OF S"  dl -> al  add,"    ENDOF
          'W' OF S"  dx -> ax  add,"    ENDOF
          'L' OF S"  edx -> eax  add,"  ENDOF
        ENDCASE emitln ;
-- ---------------------------------------------------------------
-- Subtract primary from top of stack
: _gensub ( size -- )
        S" edx -> eax  xchg," emitln
        CASE
          'B' OF S"  dl -> al  sub,"    ENDOF
          'W' OF S"  dx -> ax  sub,"    ENDOF
          'L' OF S"  edx -> eax  sub,"  ENDOF
        ENDCASE emitln ;
-- ---------------------------------------------------------------
```

ОК, я соглашусь с вами: я выдал вам множество подпрограмм с тех пор, как мы в последний раз протестировали код(types13.frt. Но вы должны признать, что каждая новая подпрограмма довольно проста и ясна. Если вам (как и мне) не нравится тестировать так много новых подпрограмм одновременно все в порядке. Вы можете заглушить подпрограммы типа Convert, Promote и SameType так как они не считывают входной поток. Вы не получите корректный код, конечно, но программа должна работать. Затем постепенно расширяйте их.

При тестировании программы не забудьте, что вы сначала должны объявить некоторые переменные а затем начать "тело" программы с "B" в верхнем регистре (для BEGIN). Вы должны обнаружить, что синтаксический анализатор обрабатывает любые аддитивные выражения. Как только все подпрограммы преобразования будет введены, вы должны увидеть, что генерируется правильный код и код для преобразования типов вставляется в нужных местах. Попробуйте смешивать переменные различных размеров а также литералы. Удостоверьтесь, что все работает правильно. Как обычно, хорошо было бы попробовать некоторые ошибочные выражения и посмотреть, как компилятор обрабатывает их.

ПОЧЕМУ ТАК МНОГО ПРОЦЕДУР?

К этому моменту вы можете подумать, что я зашел слишком далеко в смысле глубоко вложенных процедур. В этом несомненно есть большие накладные расходы. Но в моем безумии есть смысл. Как в случае с UnOp, я заглядываю вперед на время, когда мы захотим генерировать лучший код. С таким способом организации кода мы можем достичь этого без значительных изменений в программе Например, в случаях, где значение помещенное в стек не должно преобразовываться, все же лучше использовать инструкцию "вытолкнуть и сложить". Если мы решим проверять такие случаи, мы можем включить дополнительные тесты в PopAdd и PopSub не изменяя что-либо еще.

МУЛЬТИПЛИКАТИВНЫЕ ВЫРАЖЕНИЯ

Процедуры для работы с мультипликативными операторами почти такие же. Фактически, на первом уровне они почти идентичны, так что я просто покажу их здесь без особых фанфар. Первая - наша общая форма для Factor, которая включает подвыражения в скобках:

```
-- ---------------------------------------------------------------
-- Parse and translate a factor

DEFER expression ( -- typ )

: factor ( -- typ )
        Look '('  = IF  '(' match  expression  ')' match EXIT  ENDIF
        Look alpha? IF  getname load EXIT  ENDIF
        getnum  loadnumber ;
-- ---------------------------------------------------------------
-- Recognize and translate a multiply
```

```
: multiply ( typ1 -- typ2 ) '*' match  factor _popmul ;
-- -------------------------------------------------------------
-- Recognize and translate a divide
: divide   ( typ1 -- typ2 ) '/' match  factor _popdiv ;
-- -------------------------------------------------------------
-- Parse and translate a math term
: term ( -- typ )
        factor LOCAL typ
        BEGIN  Look mulop?
        WHILE  typ _push
               CASE Look
                '*' OF typ multiply TO typ ENDOF
                '/' OF typ divide   TO typ ENDOF
               ENDCASE
        REPEAT typ ;
-- -------------------------------------------------------------
```

Эти подпрограммы соответствуют аддитивным почти полностью. Как и прежде, сложность изолирована в PopMul и PopDiv. Если вам захочется протестировать программу прежде чем мы займемся ими, вы можете написать их пустые версии, аналогичные PopAdd и PopSub. И снова, код не будет корректным в данный момент, но синтаксический анализатор должен обрабатывать выражения произвольной сложности.

УМНОЖЕНИЕ

Если вы убедились, что сам синтаксический анализатор работает правильно, мы должны выяснить, что необходимо сделать для генерации правильного кода. С этого места дела становятся немного труднее так как правила более сложные.

Давайте сперва возьмем случай умножения. Эта операция аналогична "addops" в том, что оба операнда должны быть одного и того же размера. Она отличается в трех важных отношениях:

- Тип произведения обычно не такой же как тип двух операндов. Для произведения двух слов мы получаем в результате длинное слово.
- 68000 не поддерживает умножение 32 x 32, так что необходим вызов подпрограммы для программного умножения.
- Он также не поддерживает умножение 8 x 8, поэтому байтовые операнды должны быть переведены до слова.

Действия, которые мы должны выполнить, лучше всего показывает следующая таблица:

T1-> T2 v	B	W	L
B	Преобразовать EAX в W Преобразовать EDX в W MULS Result = W	Преобразовать EAX в W MULS Result = L	Преобразовать EAX в L CALL MUL32 Result = L
W	Преобразовать EDX в W MULS Result = L	MULS Result = L	Преобразовать EAX в L CALL MUL32 Result = L
L	Преобразовать EDX в L CALL MUL32 Result = L	Преобразовать EDX в L CALL MUL32 Result = L	CALL MUL32 Result = L

Эта таблица показывает действия, предпринимаемые для каждой комбинации типов операндов. Есть три вещи, на которые необходимо обратить внимание: во-первых, мы предполагаем, что существует библиотечная подпрограмма MUL32, которая выполняет 32 x 32 умножение, оставляя 32-битное (не 64) произведение. Если в процессе этого

происходит переполнение мы игнорируем его и возвращаем только младшие 32 бита.

Во-вторых, заметьте, что таблица симметрична. Наконец, обратите внимание, что произведение это всегда длинное слово, за исключением случая когда оба операнда байты. (Стоит заметить, между прочим, что это означает что результатом многих выражений будет длинное слово, нравится нам это или нет. Возможно идея перевода всех их заранее не была уж такой возмутительной, в конце концов!)

Теперь ясно, что мы должны будем генерировать различный код для 16-разрядного и 32-разрядного умножения. Для этого лучше всего иметь отдельные подпрограммы генерации кода для этих двух случаев:

```
-- ---------------------------------------------------------------------
-- Multiply secondary by primary (word)
: _genmult ( -- )  S" dx  mul," emitln ; \ dx * ax -> dx:ax
-- ---------------------------------------------------------------------
-- Multiply secondary by primary (long)
: _genlongmult ( -- )  S" edx  mul," emitln ; \ edx * eax -> edx:eax
-- ---------------------------------------------------------------------
```

Исследование кода ниже для PopMul должно убедить вас, что условия в таблице выполнены:

```
-- -----------------------------------------------------------------
-- Generate code to multiply primary by stack
: _popmul ( typ1 typ2 -- typ )
        OVER _pop  sametype >R
        R@ 'W' S" edx" convert
        R@ 'W' S" eax" convert
        R@ 'L' = IF _genlongmult
               ELSE _genmult
              ENDIF
        R> 'B' = IF  'W'  ELSE  'L'  ENDIF ;
-- -----------------------------------------------------------------
```

Как вы можете видеть, подпрограмма начинается совсем как PopAdd. Два аргумента приводятся к тому же самому типу. Два вызова Convert заботятся о случаях, когда оба операнда - байты. Сами данные переводятся до слова, но подпрограмма помнит тип чтобы назначать корректный тип результату. В заключение мы вызываем одну из двух подпрограмм генерации кода и затем назначаем тип результата. Не слишком сложно, действительно.

Я полагаю, что сейчас вы уже тестируете программу. Попробуйте все комбинации размеров операндов. (types14.frt)

ДЕЛЕНИЕ

Случай с делением совсем не так симметричен. У меня также есть для вас некоторые плохие новости:

Все современные 16-разрядные процессоры поддерживают целочисленное деление. Спецификации изготовителей описывают эту операцию как 32 x 16 бит деление, означающее, что вы можете разделить 32-разрядное делимое на 16-разрядный делитель. Вот плохая новость:

Они вам лгут!!!

Если вы не верите в это, попробуйте разделить любое большое 32-разрядное число (это означает, что оно имеет ненулевые биты в старших 16 разрядах) на целое число 1. Вы гарантированно получите исключение переполнения.

Проблема состоит в том, что эта команда в действительности требует, чтобы получаемое частное вписывалось в 16-разрядный результат. Этого не случится, если делитель достаточно большой. Когда любое число делится на единицу, частное будет конечно тем же самым, что

и делимое.

С начала времен (ну во всяком случае компьютерных) архитекторы ЦПУ предусматривали этот маленький подводный камень в схеме деления. Это обеспечивает некоторую симметрию, так как это своего рода инверсия способа каким работает умножение. Но так как единица - это совершенно допустимое (и довольно частое) число для использования в качестве делителя, делению, реализованному аппаратно, требуется некоторая помощь от программистов.

Подразумевает следующее:

- Тип частного всегда должен быть того же самого типа, что и делимое. Он независим от делителя.
- Несмотря на то, что ЦПУ поддерживает деление длинного слова, аппаратно предоставленной инструкции можно доверить только делимые байт и слово. Для делимых типа длинное слово нам необходима другая библиотечная подпрограмма, которая может возвращать длинный результат.

Это похоже на работу для другой таблицы, для суммирования требуемых действий:

T1-> T2 v	B	W	L
B	Преобразовать EAX в W Преобразовать EDX в L DIVS Result = B	Преобразовать EAX в W Преобразовать EDX в L DIVS Result = W	Преобразовать EAX в L CALL DIV32 Result = L
W	Преобразовать EDX в L DIVS Result = B	Преобразовать EDX в L DIVS Result = W	Преобразовать EAX в L CALL DIV32 Result = L
L	Преобразовать EDX в L JSR DIV32 Result = B	Преобразовать EDX в L CALL DIV32 Result = W	CALL DIV32 Result = L

(Вы можете задаться вопросом, почему необходимо выполнять 32-разрядное деление, когда делимое, скажем, всего лишь байт. Так как число битов в результате может быть только столько, сколько и в делимом, зачем беспокоиться? Причина в том, что если делитель - длинное слово и в нем есть какие-либо установленные старшие разряды, результат деления должен быть равен нулю. Мы не смогли бы получить его, если мы используем только младшее слово делителя)

Следующий код предоставляет корректную функцию для PopDiv:

```
-- ------------------------------------------------------------------------
-- Generate code to divide stack by the primary
: _popdiv ( typ1 typ2 -- typ )
        LOCALS| T2 T1 |
        T1 _pop
        T1 'L' S" edx" convert
        T1 'L' = T2 'L' =
        OR IF   T2 'L' S" eax" convert
                _genlongdiv
                'L' EXIT
        ENDIF
        T2 'W' S" eax" convert _gendiv T1 ;
-- ------------------------------------------------------------------------
```

Две подпрограммы генерации кода:

```
-- ------------------------------------------------------------------------
-- Divide secondary by primary  (word)
: _gendiv ( -- ) S" eax -> edx xchg,  dx div," emitln ;
-- ------------------------------------------------------------------------
```

```
-- Divide secondary by primary (long)
: _genlongdiv ( -- ) S" eax -> ecx mov, edx -> eax mov, cdq, ecx div," emitln ;
-- ---------------------------------------------------------------------------
```

Обратите внимание, мы предполагаем, что DIV32 оставляет результат (длинное слово) в D0.

ОК, установите новые процедуры деления. Сейчас у вас должна быть возможность генерировать код для любого вида арифметических выражений. Погоняйте ее! types14.frt)

ЗАВЕРШЕНИЕ BEGINNING TO WIND DOWN

Наконец-то, в этой главе мы узнали как работать с переменными (и литералами) различных типов. Как вы можете видеть, это не было слишком сложно. Фактически, в каком-то отношении большая часть кода выглядит даже еще проще, чем это было в более ранних программах. Только операторы умножения и деления требуют небольших размышлений и планирования.

Основная идея, которая облегчила нам жизнь, - идея преобразования процедур типа Expression в функции, возвращающие тип результата. Как только это было сделано, мы смогли сохранить ту же самую общую структуру компилятора.

Я не буду притворяться, что мы охватили каждый одиночный аспект этой проблемы. Я удобно проигнорировал беззнаковую арифметику. Из того, что мы сделали, я думаю вы можете видеть, что их включение не добавляет никаких дополнительных проблем, просто дополнительные проверки.

Я так же игнорировал логические операторы And, Or и т.д. Оказывается, их довольно легко обрабатывать. Все логические операторы - побитовые операции, так что они симметричны и, следовательно, работают в том же самом режиме, что и PopAdd. Однако, имеется одно отличие: если необходимо расширить длину слова для логической переменной, расширение должно быть сделано как число без знака. Числа с плавающей точкой, снова, являются простыми для обработки... просто еще несколько процедур, которые будут добавлены в run-time библиотеку или, возможно, инструкции для математического сопроцессора.

Возможно более важно, что я также отделил проблему контроля соответствия типов, в противоположность преобразованию. Другими словами, мы разрешили операции между переменными всех комбинаций типов. Вообще, это не будет верным... конечно вы не захотите прибавить целое число, например, к строке. Большинство языков также не позволят вам смешивать символьные и целочисленные переменные.

Снова, в действительности в этом случае нет никаких новых проблем для рассмотрения. Мы уже проверяем типы двух операндов... в основном эти проверки выполняются в процедурах типа SameType. Довольно просто включить вызов обработчика ошибок если типы двух операндов несовместимы.

В общем случае мы можем рассматривать каждый одиночный оператор как обрабатываемый отдельной процедурой, в зависимости от типа двух операндов. Это просто, хотя и утомительно, реализовать просто создав таблицу переходов с типами операндов как индексами. В Паскале эквивалентная операция включала бы вложенные операторы Case. Некторые из вызываемых процедур могли бы тогда быть простыми подпрограммами обработки ошибок, в то время как другие могли бы выполнять любые виды преобразований, необходимые нам. При добавлении нами типов, число процедур будет возрастать в геометрической прогрессии, но это все равно не неприемлемо большое число процедур.

Здесь же мы свернули такую таблицу переходов в гораздо меньшее количество процедур, просто используя симметрию и другие упрощающие правила.

ПРИВОДИТЬ ИЛИ НЕ ПРИВОДИТЬ

В случае, если до вас еще не дошло, уверен дойдет, что TINY и KISS возможно не будут строго типизированными языками, так как я разрешил автоматическое смешивание и преобразование почти любых типов. Что поднимает следующий вопрос:

Это действительно то, что мы хотим сделать?

Ответ зависит от того, какого рода язык вам нужен и как вы хотели чтобы он себя вел. Мы не обращались к проблеме того, когда разрешить или когда запретить использование операций, включающих различные типы данных. Другими словами, какова должна быть семантика нашего компилятора? Хотим ли мы выполнять автоматическое преобразование типов для всех случаев, в некоторых случаях или не выполнять совсем?

Давайте приостановимся здесь, чтобы подумать об этом немного больше. В этом нам поможет небольшой исторический обзор.

Fortran II поддерживал только два простых типа данных: Integer и Real. Он разрешал неявное преобразование типов между real и integer типами во время присваивания, но не в выражениях. Все элементы данных (включая литеральные константы) справа оператора присваивания должны были быть одинакового типа. Это довольно сильно облегчало дела... гораздо проще, чем то, что мы делали здесь.

Это было изменено в Fortran IV для поддержки "смешанной" арифметики. Если выражение имело любые real элементы, все они преобразовывались в real и само выражение было real. Для полноты, предоставлялись функции для явного преобразования из одного типа в другой, чтобы вы могли привести выражение в любой тип.

Это вело к двум вещам: код, который был проще для написания и код, который был менее эффективен. Из-за это неаккуратные программисты должны были писать выражения с простыми константами типа 0 и 1, которые компилятор должен был покорно компилировать для преобразования во время выполнения. Однако, система работала довольно хорошо, что показывало что неявное преобразование типов - Хорошая Вещъ.

Си - также слабо типизированный язык, хотя он поддерживает большее количество типов. C не будет жаловаться, если вы например попытаетесь прибавить символ к целому числу. Частично, в этом помогает соглашение Си о переводе каждого символа в число когда оно загружается или передается в списке параметров. Это совсем немного упрощает преобразование. Фактически, в подмножестве компиляторов Си, которые не поддерживают длинные или с плавающей точкой числа мы возвращаемся к нашей первой нехитрой попытке: каждая переменная получает одинаковое представление как только загружается в регистр. Жизнь становится значительно проще!

Предельным языком в направлении автоматического преобразования типов является PL/I. Этот язык поддерживает большое количество типов данных, и вы можете свободно смешивать их все. Если неявное преобразование Fortran казалось хорошим, то таковое в PL/I было бы Небесами, но оно скорее оказалось Адом! Проблема состояла в том, что с таким большим количеством типов данных должно сущестовать большое количество различных преобразований и, соответственно, большое количество правил того, как смешиваемые операнды должны преобразовываться. Эти правила стали настолько сложными, что никто не мог запомнить какие они!   Множество ошибок в программах на PL/I имели отношение к непредвиденным и нежелательным преобразованиям типов. Слишком хорошо тоже нехорошо!

Паскаль, с другой стороны, является языком, который "строго типизирован", что означает, что вы вообще не можете смешивать типы даже если они отличаются только именем, хотя они и имеют тот же самый базовый тип! Никлаус Вирт сделал Паскаль строго типизированным чтобы помочь программисту избежать проблем и эти ограничения действительно защитили многих программистов от самих себя, потому что компилятор предохранял его от глупых ошибок. Лучше находить ошибки при компиляции, чем на этапе отладки. Те же самые ограничения могут также вызвать расстройства когда вам действительно нужно смешивать типы и они заставляют бывших C-программистов лезть на стену.

Даже в этом случае, Паскаль разрешает некоторые неявные преобразования. Вы можете присвоить целое значение вещественному. Вы можете также смешивать целые и вещественные типы в выражениях типа Real. Целые числа будут автоматически приведены к вещественным, как и в Fortran. (и с теми же самыми скрытыми накладными расходами во время выполнения).

Вы не можете, однако, преобразовывать наоборот из вещественного в целое без применения явной функции преобразования Trunc. Теория здесь в том, что так как числовое значение вещественного числа обязательно будет изменено при преобразованиии (дробная часть будет потеряна), это не должно быть сделано в "секрете" от вас.

В духе строгого контроля типов Паскаль не позволит вам смешивать Char и Integer переменные без применения явных функций приведения Chr и Ord.
Turbo Pascal также включает типы Byte, Word и LongInt. Первые два в основном то же самое, что и беззнаковое целое. В Turbo они могут быть свободно смешаны с переменными типа Integer и Turbo автоматически выполнит преобразование. Однако существуют проверки времени выполнения, предохряняющие вас от переполнения или иного способа получения неправильного ответа. Заметьте, что вы все еще не можете смешивать типы Byte и Char, даже при том, что они имеют то же самое внутреннее представление.

Пределом среди строго типизированных языков является Ada, который не рязрешает никаких невных преобразований типов вообще, и также не разрешает смешанную арифметику. Позиция Jean Ichbiah в том, что преобразования стоят времени выполнения и вам нельзя позволить платить такую цену на скрытый манер. Вынуждая программиста явно запрашивать преобразование типов вы делаете более очевидным то, что здесь могут быть вовлечены затраты.

Я использовал другой язык со строгим контролем типов, небольшой восхитительный язык, названный Whimsical, от Джона Спрея. Хотя Whimsical предназначен быть языком системного программирования, он также требует каждый раз явного преобразования. В нем никогда не выполняются никакие автоматические преобразования, даже те, которые поддерживаются в Паскале.

Такой подход имеет некоторые преимущества: компилятор никогда не должен предполагать, что делать: программист всегда говорит ему точно, что он хочет. В результате, появляется почти однозначное соответствие между исходным кодом и компилированным кодом, и компилятор Джона производил очень компактный код.

С другой стороны, я иногда находил явные преобразования болезненными. Если я хочу, например, прибавить единицу к символу, или сложить ее с маской, необходимо выполнить массу преобразований. Если я сделаю это неправильно единственным сообщением об ошибке будет "Типы не совместимы". Как это случается, специфическая реализация Джоном этого языка в его компиляторе не сообщает вам точно, какие типы несовместимы... он только сообщает вам, в какой строке произошла ошибка.

Я должен признать, что большинство моих ошибок с этим компилятором были ошибками такого типа, и я потратил много времени с компилятором Whimsical, пытаясь всего-лишь выяснить в каком месте строки я ее допустил. Единственный реальный способ исправить ошибку это продолжать исправления до тех пор, пока что-нибудь не заработает.

Так что мы должны сделать в TINY и KISS? Для первого я имею ответ: TINY будет поддерживать только типы Char и Integer и мы будем использовать прием C непосредственно переводя Char в Integer. Это означает, что компилятор TINY будет значительно проще, чем то, что мы уже сделали. Необходимость преобразования типов в выражении спорна, так как ни одно из них не требуется! Так как длинное слово не будет поддерживаться, нам также будут не нужны подпрограммы MUL32 и DIV32, ни логики для выяснения когда их вызывать. Мне это нравится!

KISS, с другой стороны будет поддерживать тип Long.

Должен ли он поддерживать и знаковую и беззнаковую арифметику? Ради простоты я

предпочел бы нет. Это добавляет совсем немного сложности в преобразования типов. Даже Никлаус Вирт удалили беззнаковые числа (Cardinal) из его нового языка Оберон, с тем аргументом, что 32-разрядного целого числа в любом случае должно быть достаточно всем.

Но KISS предполагается быть языком системного программирования, что означает, что у нас должна быть возможность выполнять любые действия, которые могут быть выполнены на ассемблере. Так как 68000 поддерживает обе разновидности целых чисел, я полагаю что KISS тоже должен. Мы видели, что логические операции должны быть способны расширять целые числа беззнаковым способом, поэтому процедуры беззнакового преобразования нужны в любом случае.

ЗАКЛЮЧЕНИЕ

На этом завершается наш урок по преобразованиям типов. Жаль, что вы должны были так долго ждать его, но, надеюсь, вы чувствуете, что он того стоил.

В нескольких следующих главах мы расширим простые типы, включив поддержку массивов и указателей и посмотрим, что делать со строками. Это позволит довольно успешно завершить основную часть этой серии. После этого я дам вам новые версии компиляторов TINY и KISS, а затем мы начнем рассматривать вопросы оптимизации.

Увидимся.

15. Назад в будущее

ВВЕДЕНИЕ

Могло ли действительно пройти четыре года с тех пор, как я написал четырнадцатую главу этой серии? Действительно ли возможно, что шесть долгих лет прошли с тех пор как я начал ее? Забавно, как летит время когда вы весело его проводите, не так ли?

Я не буду тратить много времени на извинения; просто подчеркну, что это случилось, и приоритеты меняются. За четыре года, начиная с четырнадцатой главы, я сумел стать уволенным, развестись, получить нервный срыв, начал новую карьеру как писатель, начал другую как консультант, двигался, работал на две системы реального времени и вырастил четырнадцать птенцов, трех голубей, шесть опоссумов и утку. Некоторое время синтаксический анализ исходного кода был не слишком высоко в моем списке приоритетов. Не написал ни одной вещи бесплатно, только за деньги. Но я пытаюсь быть верным и понимаю и чувствую свою ответственность перед вами, читателями, закончить то, что начал. Как сказала черепаха в одной из старых историй моего сына, я возможно медленная, но я надежная. Я уверен, что есть люди, стремящиеся увидеть последнюю катушку этого фильма, и я собираюсь дать им ее. Так что, если вы один из тех, кто ждал более или менее терпеливо, что из этого получится, благодарю за ваше терпение. Я приношу извинения за задержку. Давайте продолжим.

НОВОЕ НАЧАЛО, СТАРОЕ НАПРАВЛЕНИЕ

Подобно многим другим вещам, языки программирования и стили програмирования изменяются со временем. В 1994 году кажется немного анахроничным программировать на Turbo Pascal, когда остальной мир кажется сходит с ума по C++. Также кажется немного странным программировать в классическом стиле, когда остальной мир переключился на объектно-ориентированные методы. Однако, несмотря на четырехлетнюю паузу, было бы слишком тяжело сейчас переключиться, скажем, на C++ с объектной ориентацией. Во всяком случае, Pascal все еще не только мощный язык программирования (фактически больше, чем когда либо), но это и замечательная среда для обучения. Си - известно трудный для чтения язык... он часто был обвиняем, наряду с Forth, как "язык только для записи". Когда я программирую на C++ я трачу по крайней мере 50% своего времени на борьбу с синтаксисом языка а не с концепциями. Сбивающие с толку "&" или "*" могут не только изменить функционирование программы, но также и ее правильность. Наоборот, код Паскаля

обычно совершенно ясен и прост для чтения даже если вы не знаете языка. Что вы видите, то вы почти всегда и получите, и мы можем сконцентрироваться на концепциях, а не тонкостях реализации. Я сказал в начале, что целью этой обучающей серии была не генерация самого быстрого в мире компилятора, а изучение основ технологии компиляции, с наименьшими затратами времени на борьбу с синтаксисом языка или другими аспектами реализации программного обеспечения. Наконец, так как многое из того, что мы делаем в этом курсе, составляет программное экспериментирование, важно иметь компилятор и связанную с ним среду, который компилирует быстро и без суеты. По моему мнению наиболее значимым мерилом времени при разработке программного обеспечения является скорость цикла редактирование/компиляция/тестирование. В этом отделе Turbo Pascal - король. Скорость компиляции блестяще быстрая, и продолжает становиться быстрее с каждым выпуском (как им это удается?). Несмотря на крупные усовершенствования в быстродействии компиляции C за последние годы, даже Borland-овский самый быстрый компилятор C/C++ все еще не сравним с Turbo Pascal. Далее, редактор, встроенный в его IDE, средство make, и даже их превосходный умный компоновщик, все дополняют друг друга чтобы получить замечательную среду для быстрой разработки. По всем этим причинам, я собираюсь придерживаться Паскаля в продолжении этой серии. Мы будем использовать Turbo Pascal for Windows, один из компиляторов, предоставляемый Borland Pascal with Objects, версия 7.0. Если у вас нет этого компилятора не волнуйтесь… ничего из того, что мы делаем здесь не будет рассчитано на то, что вы имеете последнюю версию. Использование Windows версии сильно помогает мне, позволяя использовать Clipboard для копирования кода из редактора компилятора в эти документы. Она также должна помочь вам по крайней мере копировать код в обратном направлении.

Я думал долго и трудно о том, надо ли представить нашему обсуждению объекты. Я большой защитник объектно-ориентированных методов для всех применений и такие методы определенно имеют свое место в технологии компиляции. Фактически, я написал несколько статей только на эти темы (ссылки 1-3). Но архитектура компилятора, основанного на объектно-ориентированных подходах, значительно отличается от архитектуры более классического компилятора, который мы строим. Кроме того, коней на переправе не меняют. Как я сказал, изменяются стили программирования. Кто знает, может быть пройдут еще шесть лет прежде чем мы закончим эти дела, и если мы будем изменять код каждый раз при изменении стиля программирования мы можем никогда не закончить.

Так что теперь, по крайней мере, я определился продолжать классический стиль в Pascal, хотя мы действительно могли бы обсуждать объекты и объектную ориентацию по ходу дела. Аналогично, целевой машиной останется семейство Motorola 68000. Из всех решений, которые буду приняты здесь, это было самым простым. Хотя я знаю, что многие из вас хотели бы видеть код для 80x86, 68000 является, вообще-то, даже более популярным как платформа для встроенных систем, и это то применение ради которого это все изначально и начиналось. Компилируя для PC, платформы MSDOS, мы должны были бы иметь дело со всеми проблемами системных вызовов DOS, форматов компоновщика DOS, файловой системы PC и аппаратными средствами и всеми другие осложнениями среды DOS. Встроенная система, с другой стороны, должна выполняться автономно и я всегда представлял, что для такого вида применений, как альтернатива ассемблеру, язык подобный KISS процветал бы. В любом случае, кто хочет иметь дело с архитектурой 80x86 если они не должны?

Одна из возможностей Turbo Pascal которую я собираюсь серьезно использовать это модули. В прошлом мы должны были делать компромиссы между размером кода, сложностью и функциональными возможностями программы. Многое из нашей работы было по природе компьютерным экспериментированием, рассматриванием только одного аспекта технологии компиляции в один момент времени. Мы делали это чтобы избежать возни с большими программами, исследуя только простые понятия. В процессе, мы заново изобретали колесо и заново программировали те же самые функции больше раз, чем я могу сосчитать. Модули Turbo предоставляют замечательный способ получить функциональность и простоту одновременно: вы пишете многократно используемый код и вызываете его в одной строке. Ваша тестовая программа остается маленькой, но она может делать мощные вещи.

Одна из возможностей модулей Turbo Pascal - их блок инициализации. Как в пакете Ada, любой код в основном блоке begin-end модуля выполняется когда программа инициализирована. Как вы увидите позже, это иногда дает нам хорошее упрощение кода. Наша процедура Init, которая была с нами начиная с Главы 1, полностью исчезает когда мы используем модули. Различные подпрограммы в Cradle, другие ключевые возможности нашего подхода, будут распределены по модулям.

Концепция модулей, конечно, ничем не отличается от модулей Си. Однако в C (и C++) интерфейс между модулями происходит через операторы include препроцессора и заголовочные файлы. Как кто-то, кто читал множество прогрограмм других людей на C, я всегда находил это довольно сбивающим с толку. Всегда кажется, что любая структура данных, о который вы бы хотели знать, находится в каком-то другом файле. Модули Turbo проще по той причине, за которую они критикуются некоторыми: интерфейсы функций и их реализации включены в тот же самый файл. В то время, когда эта организация может создать проблемы с защитой кода, она также уменьшает количество файлов наполовину, что не в два раза хуже. Связывание объектных файлов также просто, потому что компилятор Turbo заботится об этом без необходимости в файлах типа make или других механизмах.

НАЧИНАЕМ ЗАНОВО?

Четыре года назад, в Главе 14, я обещал вам, что наши дни повторного изобретения колеса и написания одних и тех же программ на каждом уроке, прошли и что с этого момента мы будем придерживаться более завершенных программ, к которым мы должны просто добавлять новые возможности. Я все еще собираюсь сдержать это обещание; это одна из основных целей использования модулей. Однако, из-за прошествия длительного времени с главы 14, естественно хотелось бы сделать по крайней мере небольшой обзор и в любом случае мы окажемся перед необходимостью сделать довольно обширные изменения кода, чтобы выполнить переход к модулям. Кроме того, если откровенно, после всего этого времени я не могу помнить всех хороших идей, которые я имел в моей голове четыре года назад. Для меня лучший способ вспомнить их - заново пройти некоторые шаги, которые привели нас к Главе 14. Так что я надеюсь, что вы поймете и смиритесь со мной когда мы возвратимся к своим корням, в некотором смысле, и перестроим ядро программы, распределяя подпрограммы по различным модулям, и вытащим сами себя назад к точке где мы были многие луны тому назад. Как всегда бывало, вы увидите все мои ошибки и смены направлений в реальном режиме времени. Пожалуйста, будьте терпеливы... мы доберемся до новых вещей раньше чем вы успеете оглянуться.

Так как в нашем новом подходе мы собираемся использовать множественные модули, мы должны обратиться к проблеме управления файлами. Если вы проследовали через все другие разделы этой обучающей серии, вы знаете, что поскольку наша программа развивается, мы заменяем старые, более простые модули на более совершенные. Это приводит нас к проблеме контроля версий. Почти обязательно будут возникать ситуации, когда мы будем перекрывать простой файл (модуль), но позднее захотим иметь его снова. Данный случай воплощен в нашей склонности к использованию односимвольных имен переменных, ключевых слов и т.д. для проверки основных понятий не захлебываясь в деталях лексического анализатора. Благодаря использованию модулей мы будем намного меньше делать это в будущем. Однако, я не только предполагаю, но я уверен, что мы будем должны сохранять некоторые старые версии файлов для специальных целей, даже при том, что они заменяются более новыми, более совершенными.

Для решения этой проблемы я предлагаю, чтобы вы создали различные каталоги с различными версиями модулей. Если мы сделаем это правильно, код в каждом каталоге останется само-непротиворечивым. Я в порядке эксперимента создал четыре каталога: SINGLE (для односимвольных экспериментов), MULTI (для, конечно, многосимвольной версии), TINY и KISS.

Достаточно сказано о философии и деталях. Давайте продолжим восстановление программы.

МОДУЛЬ INPUT

Ключевой концепцией, которую мы использовали начиная с первого дня, была идея входного потока с одним предсказывающим символом. Все подпрограммы синтаксического анализа проверяют этот символ, не изменяя его, чтобы решить, что они должны делать дальше. (Сравните этот подход с подходом C/Unix, использующим getchar и unget, и я думаю вы согласитесь, что наш подход проще). Мы начнем нашу экскурсию в будущее перенеся эту концепцию в нашу новую модульную организацию. Первый модуль, соответствующе названный Input, показан ниже:

```
-- -------------------------------------------------------------
-- input
-- -------------------------------------------------------------
0 VALUE Look -- Lookahead character
-- -------------------------------------------------------------
-- Read new character from input stream
: getchar ( -- ) EKEY TO Look ;
-- -------------------------------------------------------------
-- Unit initialization
        getchar
-- -------------------------------------------------------------
```

Как вы можете видеть, здесь нет ничего очень заумного и конечно ничего сложного, так как он состоит только из одной процедуры. Но мы уже можем видеть как использование модулей дает нам преимущества. Обратите внимание на выполнимый код в блоке инициализации. Этот код "запускает помпу" входного потока для нас, нечто такое мы всегда делали раньше вставляя вызовы GetChar в процедуру Init. На этот раз вызов происходит без каких-либо специальных обращений к ней с нашей стороны, за исключением самого модуля. Как я предсказывал ранее, этот механизм сделает нашу жизнь в будущем значительно проще. Я полагаю это одна из наиболее полезных возможностей Turbo Pascal и я буду сильно на нее полагаться.

Скопируйте этот модуль в IDE вашего компилятора и откомпилируйте его. Чтобы проверить программу, конечно, нам всегда нужна основная программа. Я использовал следующую, действительно сложную тестовую программу, которую позже мы разовьем в главную для нашего компилятора:

```
-- -------------------------------------------------------------
NEEDS -input
: main1 ( -- ) Look EMIT ;
-- -------------------------------------------------------------
```

Обратите внимание на использование предоставляемого Borland модуля WinCRT. Этот модуль необходим, если вы предполагаете использовать стандартные подпрограммы ввода/вывода Паскаля Read, ReadLn, Write и WriteLn, которые мы конечно предполагаем использовать. Если вы забудете включить этот модуль в раздел "uses" вы получите действительно причудливое и непонятное сообщение во время выполнения.

Заметьте также, что мы можем обращаться к предсказывающему символу даже при том, что он не объявлен в основной программе. Все переменные, объявленные в разделе interface модуля, являются глобальными, но они скрыты от любопытных глаз; в какой-то степени мы получаем чуточку сокрытия информации. Конечно, если бы мы писали в объектно-ориентированном стиле, мы не должны были бы позволять обращаться к внутренним переменным модуля снаружи. Но хотя модули Turbo имеют много общего с объектами, мы не собираемся здесь реализовывать объектно ориентированный дизайн или код, так что мы используем Look соответствующее.

Продолжим и сохраним тестовую программу как Main1.frt.(main1.frt) Чтобы сделать жизнь проще когда файлов будет становиться все больше и больше, вам возможно захотелось бы

использовать возможность объявить этот файл как Primary файл компилятора. Таким способом вы можете выполнять программу из любого файла. Иначе, если вы нажимете Cntl-F9 для компиляции и выполнения одного из модулей, вы получите сообщение об ошибке. Вы устанавливаете primary-файл используя главное подменю "Compile" в Turbo IDE.

Я тороплюсь отметить, как я делал раньше, что функционально модуль Input является, и всегда был, макетом настоящей версии. В серийной версии компилятора входной поток будет, конечно, скорее исходить из файла, а не клавиатуры. И это почти обязательно включает буферизацию строки, по крайней мере, и более вероятно, довольно большой текстовый буфер для поддержки эффективного дискового ввода/вывода. Хорошая сторона модулей в том, что как и с объектами мы можем делать код модуля таким простым или таким сложным как нам угодно. До тех пор, пока интерфейс, встроенный в общедоступные процедуры и предсказывающий символ не изменяются, остальная часть программы абсолютно незатрагивается. И так как модули компилируются, а не просто включаются, время необходимое для связывания их вместе практически равно нулю. Снова, результат таков, что мы можем получить все преимущества сложной реализации без необходимости возиться с кодом как лишним багажом.

В следующих главах я предполагаю предоставить полноценный IDE для компилятора KISS используя настоящее Windows приложение, сгенерированное с помощью среды разработки Borland OWL. Сейчас, тем не менее, мы удовлетворим мое первое правило: Делать Это Проще.

МОДУЛЬ OUTPUT

Конечно, каждая приличная программа должна выводить результат и наша не исключение. Наши подпрограммы вывода включают функции Emit. Код для соответствующего модуля показан дальше:

```
-- -------------------------------------------------------------
-- output
-- -------------------------------------------------------------
-- Emit an instruction
: emits  ( c-addr u -- ) Tab EMIT TYPE ;
-- -------------------------------------------------------------
-- Emit an instruction, followed by a newline
: emitln ( c-addr u -- ) CR emits ;
-- -------------------------------------------------------------
```

(Заметьте, что этот модуль не имеет раздела инициализации, так что он не требует блока begin.)

Проверьте этот модуль с помощью следующей основной программы:(test.frt

```
-- -------------------------------------------------------------

NEEDS -input
NEEDS -output
: test ( -- ) CR ." MAIN:" S" Hello, world!" emitln ;
-- -------------------------------------------------------------
```

Увидели ли вы что-либо, что удивило вас? Вы возможно были удивлены видеть, что вам было необходимо что-то набрать даже хотя основная программа не требует никакого ввода. Дело в разделе инициализации модуля Input, который все еще требует поместить что-либо в предсказывающий символ. Жаль, нет никакого способа выйти из этого, или скорее, мы не хотим выходить. За исключением простых тестовых случаев, как этот, нам всегда будет необходим допустимый предсказывающий символ, так что самое лучшее, что мы можем сделать с этой "проблемой" это... ничего.

Возможно более удивительно то что символ TAB не имеет никакого эффекта; наша строка "инструкций" начинается с первой колонки так же как и фальшивая метка... Правильно:

WinCRT не поддерживает табуляцию. У нас проблема.

Есть несколько способов, с помощью которых мы можем решить эту проблему. Один из вариантов того что мы можем сделать - просто игнорировать ее. Каждый ассемблер, который я когда либо использовал, резервируют колонку 1 для меток и взбунтуется когда увидит, что в ней начинаются инструкции. Так что, по крайней мере, мы должны сдвинуть инструкции на одну колонку чтобы сделать ассемблер счастливым. Это достаточно просто сделать: просто измените в процедуре Emit строку:

```
Write(TAB, s);
```

на

```
Write(' ', s);
```

Я должен признать что сталкивался с этой проблемой раньше и находил себя меняющим свое мнение так часто как хамелеон меняет цвет. Для наших целей, 99% которых будет проверка выходного кода при выводе на CRT, было бы хорошо видеть аккуратно сгруппированный "объектный" код. Строка:

```
SUB1:           MOVE #4,D0
```

просто выглядит более опрятно, чем отличающийся, но функционально идентичный код:

```
SUB1:
 MOVE #4,D0
```

В тестовой версии моего кода я включил более сложную версию процедуры PostLabel, которая позволяет избежать размещения меток на раздельных строках, задерживая печать метки чтобы она оказалась на той же самой строке, что и связанная инструкция. Не позднее чем час назад, моя версия модуля Output предоставляла полную поддержку табуляции с использованием внутренней переменной счетчика столбцов и подпрограммы для ее управления. Я имел некоторый довольно изящный код для поддержки механизма табуляции с минимальным увеличением кода. Было ужасное искушение показать вам эту "красивую" версию, единственно чтобы покрасоваться элегантностью.

Однако, код "элегантной" версии был значительно более сложным и большим. После этого у меня появилась вторая мысль. Несмотря на наше желание видеть красивый вывод, неизбежный факт то, что две версии MAIN: фрагменты кода, показанные выше функционально идентичны; ассемблер, который является конечной целью кода, не интересует какую версию он получает, за исключением того, что красивая версия будет содержать больше символов, следовательно будет использовать больше дискового пространства и дольше ассемблироваться. Но красивая версия не только генерирует больше кода, но дает больший выходной файл, с гораздо большим количество пустых символов чем минимально необходимо. Когда вы посмотрите на это с такой стороны, то не трудно будет решить какой подход использовать, не так ли?

То что наконец решило для меня этот вопрос было напоминанием считаться с моей первой заповедью: KISS. Хотя я был довольно горд всеми своими изящными приемчиками для реализации табуляции, я напомнил себе, что перефразируя сенатора Барри Голдватера, элегантность в поисках сложности не является достоинством. Другой мудрый человек однажды написал: "Любой идиот может разработать Роллс-Ройс. Требуется гений, чтобы разработать VW". Так что изящная, дружественная табуляции версия Output в прошлом, и то, что вы видите, это простая компактная VW версия.

МОДУЛЬ ERROR

Наш следующий набор подпрограмм обрабатывает ошибки. Чтобы освежить вашу память мы возьмем подход, заданный Borland в Turbo Pascal - останавливаться на первой ошибке. Это не только значительно упрощает наш код, полностью устраняя назойливую проблему

восстановления после ошибок, но это также имеет намного больший смысл, по моему мнению, в интерактивной среде. Я знаю, что это может быть крайней позицией, но я считаю практику сообщать обо всех ошибках в программе анахронизмом, пережитком со времен пакетной обработки. Пришло время прекратить такую практику. Так вот.

В нашем оригинальном Cradle мы имели две процедуры обработки ошибок: Error, которая не останавливалась, и Abort, которая останавливалась. Но я не думаю, что мы когда-либо найдем применение процедуре, которая не останавливается, так что в новом, тощем и скромном модуле Errors, показанном ниже, процедура Error занимает место Abort.

```
-- ----------------------------------------------------------------------
-- Errors
-- ----------------------------------------------------------------------
-- Write error message and halt
: errors ( c-addr u -- ) CR ^G EMIT ." Error: " TYPE '.' EMIT  ABORT ;
-- ----------------------------------------------------------------------
-- Write "" expected
: expected ( c-addr u -- ) S"  expected" $+ errors ;
-- ----------------------------------------------------------------------
```

Как обычно, вот программа для проверки:test2.frt

```
-- ----------------------------------------------------------------------
NEEDS -input
NEEDS -output
NEEDS -errors
: test2 ( -- ) S" Integer" expected ;
-- ---------------------------------------------------------------
```

Вы заметили, что строка "uses" в нашей основной программе становится длиннее? Это нормально. В конечной версии основная программа будет вызывать процедуры только из нашего синтаксического анализатора, так что раздел uses будет иметь только пару записей. Но сейчас возможно самое лучшее включить все модули, чтобы мы могли протестировать процедуры в них.

ЛЕКСИЧЕСКИЙ И СИНТАКСИЧЕСКИЙ АНАЛИЗ

Классическая архитектура компилятора основана на отдельных модулях для лексического анализатора, который предоставляет лексемы языка, и синтаксического анализатора, который пытается определить смысл токенов как синтаксических элементов. Если вы еще не забыли что мы делали в более ранних главах, вы вспомните, что мы не делали ничего подобного. Поскольку мы используем предсказывающий синтаксический анализатор, мы можем почти всегда сказать, какой элемент языка следует дальше, всего-лишь исследуя предсказывающий символ. Следовательно, нам не нужно предварительно выбирать токен, как делал бы сканер.

Но даже хотя здесь и нет функциональной процедуры, названной "Scanner", все еще имеет смысл отделить функции лексического анализа от функций синтаксического анализа. Так что я создал еще два модуля, названных, достаточно удивительно, Scanner и Parser. Модуль Scanner содержит все подпрограммы, известные как распознаватели. Некоторые из них, такие как IsAlpha, являются чисто булевыми подпрограммами, которые оперируют только предсказывающим символом. Другие подпрограммы собирают токены, такие как идентификаторы и числовые константы. Модуль Parser будет содержать все подпрограммы, составляющие синтаксический анализатор с рекурсивным спуском. Общим правилом должно быть то, что модуль Parser содержит всю специфическую для языка информацию; другими словами, синтаксис языка должен полностью содержаться в Parser. В идеальном мире это правило должно быть верным в той степени, что мы можем изменять компилятор для компиляции различных языков просто заменяя единственный модуль Parser.

На практике, дела почти никогда не бывают такими чистыми. Все есть небольшая "утечка" синтаксических правил также и в сканер. К примеру, правила составления допустимого

идентификатора или константы могут меняться от языка к языку. В некоторых языках правила о комментариях разрешают им быть отфильтрованными в сканере, в то время как другие не разрешают. Так что на практике оба модуля вероятно придут к тому, что будут иметь языко-зависимые компоненты, но изменения, необходимые для сканнера, должны быть относительно тривиальными.

Теперь вспомните, что мы использовали две версии подпрограмм лексического анализатора: одна, которая поддерживала только односимвольные токены, которую мы использовали в ряде наших тестов, и другая, которая предоставляет полную поддержку многосимвольных токенов. Теперь, когда мы разделяем нашу программу на модули, я не ожидаю многого от использования односимвольной версии, но не потребуется многого, чтобы предусмотреть их обе. Я создал две версии модуля Scanner. Первая, названная Scanner1, содержит односимвольную версию подпрограмм распознавания:

```
-- -------------------------------------------------------------
-- Scanner1
-- -------------------------------------------------------------
NEEDS -input
NEEDS -errors
-- -------------------------------------------------------------
: alpha?   ( char -- tf )  >UPC 'A' 'Z' 1+ WITHIN ;             -- recognize an
alpha character
: digit?   ( char -- tf )  '0' '9' 1+ WITHIN ;                  -- recognize a
decimal digit
: alnum?   ( char -- tf )  DUP alpha? SWAP digit? OR ;          -- recognize
alphanumeric
: addop?   ( char -- tf )  DUP '+' =  SWAP '-' =  OR ;          -- test for AddOp
: mulop?   ( char -- tf )  DUP '*' =  SWAP '/' =  OR ;          -- test for MulOp
-- -------------------------------------------------------------
-- match a specific input character
: match ( char -- )
        DUP Look = IF  DROP getchar
                 ELSE  S" `" ROT CHAR-APPEND &' CHAR-APPEND expected
                ENDIF ;
-- -------------------------------------------------------------
-- get an identifier
: getname ( -- char )
        Look alpha? 0= IF  S" Name" expected  ENDIF
        Look >UPC
        getchar ;
-- -------------------------------------------------------------
-- get a number
: getnum ( -- char )
        Look digit? 0= IF  S" Integer" expected  ENDIF
        Look
        getchar ;
-- -------------------------------------------------------------
```

Следующий фрагмент кода основной программы обеспечивает хорошую проверку лексического анализатора. Для краткости я включил здесь только выполнимый код; остальное тоже самое. Не забудьте, тем не менее, добавить имя Scanner1 в раздел "uses"(main2.frt):

```
-- ---------------------
NEEDS -scanner1
: main2 ( -- )
        getname    EMIT
        '=' match
        getnumber  EMIT
        '+' match
        getname CR EMIT ;
-- ---------------------
```

Этот код распознает все предложения вида:

```
x=0+y
```

где x и y могут быть любыми односимвольными именами переменных и 0 любой цифрой. Код должен отбросить все другие предложения и выдать осмысленное сообщение об ошибке.

Если это произошло, тогда вы в хорошей форме и мы можем продолжать.

МОДУЛЬ SCANNER

Следующая, и намного более важная, версия лексического анализатора, та которая обрабатывает многосимвольные токены, которые должны иметь все настоящие языки. Только две функции, GetName и GetNumber отличаются в этих двух модулях, но только чтобы убедиться, что здесь нет никаких ошибок, я воспроизвел здесь весь модуль. Это модуль Scanner:

```
-- --------------------------------------------------------------
-- Scanner
-- --------------------------------------------------------------
NEEDS -input
NEEDS -errors
-- --------------------------------------------------------------
: alpha?   ( char -- tf )  >UPC 'A' 'Z' 1+ WITHIN ;             -- recognize an
alpha character
: digit?   ( char -- tf )  '0' '9' 1+ WITHIN ;                  -- recognize a
decimal digit
: alnum?   ( char -- tf )  DUP alpha? SWAP digit? OR ;          -- recognize
alphanumeric
: addop?   ( char -- tf )  DUP '+' =  SWAP '-' =  OR ;          -- test for AddOp
: mulop?   ( char -- tf )  DUP '*' =  SWAP '/' =  OR ;          -- test for MulOp
-- --------------------------------------------------------------
-- match a specific input character
: match ( char -- )
        DUP Look = IF  DROP getchar
                 ELSE  S" `" ROT CHAR-APPEND &' CHAR-APPEND expected
                ENDIF ;
-- --------------------------------------------------------------
CREATE token$  0 C, #256 CHARS ALLOT    -- unencoded token

: char+!  ( c addr -- ) DUP >S COUNT + C!  1 S> C+! ;
-- --------------------------------------------------------------
-- get an identifier
: getname ( -- c-addr u )
        Look alpha? 0= IF  S" Name" expected  ENDIF
        token$ C0!
        BEGIN   Look alnum?
        WHILE   Look >UPC token$ char+!  getchar
        REPEAT
        token$ COUNT ;
-- --------------------------------------------------------------
-- get a number
: getnum ( -- c-addr u )
        Look digit? 0= IF  S" Integer" expected  ENDIF
        token$ C0!
        BEGIN   Look digit?
        WHILE   Look token$ char+!  getchar
        REPEAT
        token$ COUNT ;
-- --------------------------------------------------------------
```

Таже самая тестовая программа проверит также и этот сканер.

```
-- ----------------------
NEEDS -scanner
: main3 ( -- )
```

```
        getname    TYPE
        '=' match
        getnumber  TYPE
        '+' match
        getname CR TYPE ;
-- ----------------------
```

Просто измените раздел "uses" для использования Scanner вместо Scanner1. Теперь у вас должна быть возможность набирать многосимвольные имена и числа.main3.frt)

РЕШЕНИЯ, РЕШЕНИЯ

Несмотря на относительную простоту обоих сканеров, много идей вошло в них и много решений было сделано. Я хотел бы поделиться этими мыслями с вами сейчас чтобы вы могли принимать свои собственные решения, соответствующие вашему приложению. Сначала заметьте, что обе версии GetName переводят входные символы в верхний регистр. Очевидно, здесь было принято проектное решение, и это один из тех случаев, когда синтаксис языка распределяется по лексическому анализатору. В языке Си регистр символов имеет значение. Для такого языка мы, очевидно, не сможем преобразовывать символы в верхний регистр. Дизайн, который я использую, предполагает язык, подобный Pascal, в котором регистр символов не имеет значения. Для таких языков проще идти вперед и преобразовывать все идентификаторы в верхний регистр в лексическом анализаторе, так что мы не должны волноваться позднее, когда вы сравниваем строки.

Мы могли бы даже пойти дальше и преобразовывать символы в верхний регистр прямо когда они заходят, в GetChar. Этот метод также работает, и я использовал его в прошлом, но он слишком ограничивающий. В частности, он также преобразует символы, которые могут быть частью строк в кавычках, что не является хорошей идеей. Так что если вы вообще собираетесь преобразовывать символы в верхний регистр, GetName подходящее место сделать это.

Обратите внимание, что функция GetNumber в этом сканере возвращает строку, так же как и GetName. Это одна из тех вещей, относительно которых я колебался почти что ежедневно, и последнее колебание было всего десять минут назад. Альтернативный подход и подход, который я использовал много раз в прошлых главах возвращает целочисленный результат.

Оба подхода имеют свои преимущества. Так как мы выбираем число, метод, который немедленно приходит на ум - возвращать его как целое число. Но имейте ввиду, что возможно число будет использоваться в операторе вывода который возвращает его во внешний мир. Кто-то, или мы или код, скрытый внутри оператора вывода, окажется перед необходимостью снова преобразовывать число обратно в строку. Turbo Pascal включает такие подпрограммы преобразования строк, но зачем использовать их если мы не должны? Зачем преобразовывать число из строковой в целочисленную форму только для того, чтобы конвертировать его обратно в генераторе кода, всего несколько операторов спустя?

Кроме того, как вы скоро увидите, нам будет необходимо временное место для хранения токена, который мы извлекли. Если мы обрабатываем числа в их строковой форме, мы можем сохранять значение и переменной и числа в той же самой строке. В противном случае мы должны создать вторую, целочисленную переменную.

С другой стороны, мы обнаружим, что обработка числа как строки фактически уничтожает любую возможность дальнейшей оптимизации. Когда мы доберемся до точки, где мы начнем заниматься генерацией кода, мы столкнемся со случаями, в которых мы выполняем вычисления с константами. Для таких случаев действительно глупо генерировать код, выполняющий арифметику с константами во время выполнения. Гораздо лучше позволить синтаксическому анализатору выполнять арифметику во время компиляции и просто кодировать результат. Чтобы сделать это нам необходимо сохранять константы как целые числа а не строки.

В конце концов обратно к строковому подходу меня склонило энергичное тестирование

KISS, плюс напоминание самому себе, что мы тщательно избегаем проблем эффективности кода. Одна из вещей, которые заставляют нашу нехитрую схему синтаксического анализа работать, без сложностей "настоящего" компилятора, это то, что мы прямо сказали что мы не затрагиваем эффективность кода. Это дает нам массу свободы выполнять работу простейшим путем а не эффективнейшим, и эту свободу мы должны стремиться не потерять, не смотря на призывы к эффективности звучащие в наших ушах. В дополнение к тому, что я большой сторонник философии KISS я также защитник "ленивого программирования", что в этом контексте означает не программировать что-либо пока вы не нуждаетесь в этом. Как говорит П. Дж. Плоджер "никогда не откладывайте на завтра то, что вы можете отложить насовсем". Годами писался код, предоставлявший возможности, которые не были никогда использованы. Я научился этому сам на горьком опыте. Так что вывод таков: мы не будем конвертировать в целое число потому, что это нам не нужно.

Для тех из вас, что все еще думает, что нам может быть нужна целочисленная версия (и действительно она может нам понадобиться), вот она:

```
-- --------------------------------------------------------------
-- get a number (integer version)
: getnum ( -- val )
        newline
        Look digit? 0= IF  S" Integer" expected  ENDIF
        0 >R
        BEGIN   Look digit?
        WHILE   R> #10 *  Look '0' - +  >R
                getchar
        REPEAT  '#' TO token
        R> ;
-- --------------------------------------------------------------
```

Вы могли бы отложить ее, как я предполагаю, на черный день.

СИНТАКСИЧЕСКИЙ АНАЛИЗ

К этому моменту мы распределили все подпрограммы, составляющие наш Cradle, в модули, которые мы можем вытаскивать когда они необходимы. Очевидно, они будут развиваться дальше когда мы снова продолжим процесс восстановления, но большая часть их содержимого и несомненно архитектура, которую они подразумевают, определена. Остается воплотить синтаксис языка в модуль синтаксического анализа. Мы не будем делать многого из этого в этой главе, но я хочу сделать немного просто чтобы оставить вас с хорошим чувством, что мы все еще знаем что делаем. Так что прежде, чем мы продолжим давай сгенерируем синтаксический анализатор достаточный только для обработки одиночного показателя в выражении. В процессе мы также обнаружим, что по необходимости создали также модуль генератора кода.

Помните самую первую главу этой серии? Мы считывали целочисленное значение, скажем n, и генерировали код для его загрузки в регистр D0 через move:

```
       n d# -> eax  mov,
```

Немного погодя, мы повторили этот процесс для переменной,

```
        X dword-ptr -> eax  mov,
```

а затем для показателя, который может быть и константой и переменной. В память о прошлом, давайте повторим этот процесс Определите следующий новый модуль:

```
-- --------------------------------------------------------------
-- Parser
-- --------------------------------------------------------------
NEEDS -input
NEEDS -scanner
NEEDS -codegen
```

```
-- ---------------------------------------------------------------
-- Parse and translate a factor
: factor ( -- ) getnumber loadconstant ;
-- ---------------------------------------------------------------
```

Как вы можете видеть, этот модуль вызывает процедуру LoadConstant, которая фактически выполняет вывод ассемблерного кода. Модуль также использует новый модуль CodeGen. Этот шаг представляет последнее главное изменение в нашей архитектуре с более ранних глав: перемещение машино-зависимого кода в отдельный модуль. Если я дойду до конца, вне CodeGen не будет ни одной строчки кода, которая указывала бы на то, что мы нацелены на процессор 68000. И это то место, которое показывает, что моя цель достижима.

Для тех из вас, кто желает, чтобы я использовал архитектуру 80x86 (или любую другую) вместо 68000, вот мой ответ: просто замените CodeGen на подходящий для вашего ЦПУ.

Пока наш генератор кода содержит только одну процедуру. Вот этот модуль:

```
-- ---------------------------------------------------------------
-- CodeGen
-- ---------------------------------------------------------------
NEEDS -output
-- ---------------------------------------------------------------
-- Load the primary register with a constant
: LoadConstant ( c-addr u -- ) S"  d# -> eax  mov," $+ emitln ;
-- ---------------------------------------------------------------
```

Скопируйте и откомпилируйте этот модуль и выполните следующую основную программу: (main4.frt):

```
-- ---------------------------------------------------------------
NEEDS -input
NEEDS -output
NEEDS -errors
NEEDS -scanner
NEEDS -parser
: main4 ( -- ) factor ;
-- ---------------------------------------------------------------
```

Вот он, сгенерированный код, такой как мы и надеялись.

Теперь, я надеюсь, вы можете начать видеть преимущества модульной архитектуры нашего нового проекта. Здесь мы имеем основную программу длиной всего пять строк. Это все, что нам нужно видеть, если мы не захотим видеть больше. И пока все эти модули сидят здесь терпеливо ожидая когда смогут послужить нам. Наше преимущество в том, что мы имеем простой и короткий код, но мощных союзников. Что остается сделать, это расширить модули до уровня возможностей более ранних глав. Мы сделаем это в следующей главе, но прежде, чем я закончу, давайте закончим синтаксический анализ показателя только для того, чтобы убедить себя, что мы знаем как. Конечная версия CodeGen включает новую процедуру LoadVariable:

```
-- ---------------------------------------------------------------
-- CodeGen
-- ---------------------------------------------------------------
NEEDS -output
-- ---------------------------------------------------------------
-- Load the primary register with a constant
: loadconstant ( c-addr u -- )
        S"  d# -> eax  mov," $+ emitln ;
-- ---------------------------------------------------------------
-- Load a variable to the primary register
: loadvariable  ( c-addr u -- )
        S"  dword-ptr -> eax  mov," $+ emitln ;
```

```
-- --------------------------------------------------------------
```

Сам модуль Parser не изменяется, но мы имеем более сложную версию процедуры Factor:

```
-- --------------------------------------------------------------
-- Parse and translate a factor
: factor ( -- )
        Look digit? IF  getnumber loadconstant  EXIT  ENDIF
        Look alpha? IF  getname   loadvariable  EXIT  ENDIF
        S" Unrecognized character `" Look CHAR-APPEND
           &' CHAR-APPEND errors ;
-- --------------------------------------------------------------
```

Теперь, без изменений основной программы, вы должны обнаружить, что программа обрабатывает и переменный и постоянный показатель. К этому моменту наша архитектура почти завершена; у нас есть модули, выполняющие всю грязную работу и достаточно кода в синтаксическом анализаторе и генераторе кода чтобы продемонстрировать что все работает. Остается расширить модули которые мы определили, в особенности синтаксический анализатор и генератор кода, для поддержки более сложных синтаксических элементов, которые составляют настоящий язык. Так как мы делали это много раз прежде в предыдущих главах, не должно занять у нас много времени вернуться назад к тому месту, где мы были до долгого перерыва. Мы продолжим этот процесс в Главе 16, которая скоро появится. Увидимся.

ССЫЛКИ

1. Crenshaw, J.W., "Object-Oriented Design of Assemblers and Compilers," Proc. Software Development '91 Conference, Miller Freeman, San Francisco, CA, February 1991, pp. 143-155.
2. Crenshaw, J.W., "A Perfect Marriage," Computer Language, Volume 8, #6, June 1991, pp. 44-55.
3. Crenshaw, J.W., "Syntax-Driven Object-Oriented Design," Proc. 1991 Embedded Systems Conference, Miller Freeman, San Francisco, CA, September 1991, pp. 45-60.

16. Конструирование модулей

ВВЕДЕНИЕ

Эта обучающая серия обещает стать возможно одной из самых долгоиграющих мини-серий в истории, конкурирующей только с задержкой на Томе IV Кнута. Начатая в 1988, эта серия вошла в четырехлетнюю паузу в 1990, когда "заботы мира сего", изменения в приоритетах и интересах и необходимость зарабатывать на жизнь казалось забросили ее после Главы 14. Долготерпевшие из вас были наконец вознаграждены весной прошлого года долгожданной Главой 15. В ней я начал попытку поставить серию обратно на рельсы и по ходу дела сделать ее проще для достижения цели, которая состоит в том, чтобы обеспечить вас не только достаточным пониманием трудных тем теории компиляции, но также достаточными инструментами в виде фиксированных подпрограмм и концепций, так чтобы вы были способны продолжать самостоятельно и стали достаточно опытными для того, чтобы создавать свои собственные синтаксические анализаторы и трансляторы. Из-за этой длинной паузы я подумал что следует вернуться назад и повторно рассмотреть концепции, которые мы до этого охватили а также заново сделать некоторые части программы. В прошлом мы никогда сильно не касались разработки программных инструментов промышленного качества... в конце концов я пытался обучать (и обучаться) концепциям, а не промышленной

практике. Чтобы сделать это я старался давать вам не законченные компиляторы и анализаторы, а только те отрывки кода, которые иллюстрировали частные случаи, которые мы рассматривали в текущий момент.

Я все еще верю, что это хороший способ изучения любого вопроса; никто не захочет вносить в изменения в программу в 100,000 строк только для того чтобы попробовать новую идею. Но идея работы с обрывками кода а не полными программами также имеет свои недостатки из-за которых мы писали те же самые фрагменты кода много раз. Хотя было полностью доказано, что повторение является хорошим способом обучения новым идеям, также правда и то, что оно может быть не слишком хорошей вещью. Ко времени, когда я завершил Главу 14, я казалось достиг пределов своих способностей манипулировать множеством файлов и множественными версиями тех же самых программ. Кто знает, может быть это одна из причин, по которым я кажется выдохся в то время.

К счастью, более поздние версии Borland Turbo Pascal позволяют нам получить и съесть свой кусок пирога. Используя их концепцию раздельно компилируемых модулей мы все еще можем писать маленькие подпрограммы и функции и сохранять наши основные и тестовые программы маленькими и простыми. Но, однажды написанный, код в модулях Паскаля будет всегда там для нашего использования и его связывание абсолютно безболезненно и прозрачно.

Так как к настоящему времени большинство из вас программируют на C или C++, я знаю, что вы подумаете: Borland с их Turbo Pascal конечно не изобретали понятие раздельно компилируемых модулей. И, конечно, вы правы. Но если вы не использовали TP в последнее время или когда либо, вы можете не понять насколько безболезненный весь этот процесс. Даже в C или C++ вы все еще должны формировать make файл, или вручную, или сообщая компилятору как это сделать. Вы должны также перечислить, используя утверждение "extern" или заголовочные файлы, функции, которые вы хотите импортировать. В TP вы не должны даже делать этого. Вам необходимы только имена модулей, которые вы желаете использовать, и все их процедуры автоматически становятся доступны.

У меня нет намерения заниматься здесь дебатами на тему войн языков, так что я не буду затрагивать эту тему в дальнейшем. Даже я больше не использую Pascal в своей работе... я использую C на работе и C++ для своих статей в Embedded Systems Programming и других журналах. Поверьте мне, когда я намеревался возродить эту серию, я думал долго и интенсивно о переключении и языка и целевой системы на те, которые мы все используем в эти дни, C/C++ и архитектуру PC и возможно также и объектно-ориентированные методы. В конце концов я понял, что это вызовет больше беспорядка, чем сам перерыв. И в конце концов, Pascal все еще остается одним из лучших возможных языков для обучения, не говоря о промышленном программировании. Наконец, TP все еще компилирует на скорости света, гораздо быстрее чем конкурирующие C/C++ компиляторы. А интеллектуальный компоновщик Borland, использованный в TP но не в их продуктах C++ не имеет аналогов. Кроме того, что он намного быстрее, чем Microsoft-совместимые компоновщики, Borland-овский интеллектульный компоновщик отберет неиспользуемые процедуры и элементы данных даже вплоть до вырезания их из определенных объектов если они не нужны. Один из редких моментов нашей жизни, когда мы не должны идти на компромисс между полнотой и эффективностью. Когда мы пишем модуль TP мы можем сделать его настолько полным как нам нравится, включая любые функции и элементы данных которые, как мы думаем, могут нам когда-либо понадобиться, уверенные, что это не будет создавать ненужного раздутия кода в откомпилированной выполнимой программе.

Главное в действительности в следующем: используя механизм модулей TP мы можем иметь все преимущества и удобства написания маленьких, на вид автономных тестовых программ, без необходимости постоянно переписывать необходимые функции поддержки. Однажды написанные, модули TP сидят там, тихонько ожидая возможности выполнить свой долг и дать нам необходимую поддержку, когда будет необходимо.

Используя этот принцип, в Главе 15 я намеревался минимизировать нашу тенденцию

заново изобретать колесо, организуя наш код в отдельные модули Turbo Pascal, каждый из которых содержит различные части компилятора. Мы завершили со следующими модулями:

- Input
- Output
- Errors
- Scanner
- Parser
- CodeGen

Каждый из этих модулей обслуживает разные функции и изолирует специфические области функциональных возможностей. Модули Input и Output, как подразумевают их имена, обеспечивают ввод/вывод символьного потока и важнейший предсказывающий символ, на котором основан наш предсказывающий синтаксический анализатор. Модуль Errors конечно обеспечивает стандартную обработку ошибок. Модуль Scanner содержит все наши булевы функции типа IsAlpha и подпрограммы GetName и GetNumber, которые обрабатывают многосимвольные токены.

Два модуля, с которыми мы будем в основном работать и те, которые больше всего представляют индивидуальность нашего компилятора - это Parser и CodeGen. Теоретически модуль Parser должен изолировать все аспекты компилятора, которые зависят от синтаксиса компилируемого языка (хотя, как мы видели последний раз, небольшое количество этого синтаксиса перетекает в Scanner). Аналогично, модуль генератора кода, CodeGen, содержит весь код, зависящий от целевой машины. В этой главе мы продолжим разработку функций в этих двух важнейших модулях.

СОВСЕМ КАК КЛАССИЧЕСКИЙ?

Прежде чем мы продолжим, однако, я думаю что должен разъяснить связи между модулями и функциональные возможности этих модулей. Те из вас, кто знаком с теорией компиляции как обучавшиеся в университетах, конечно распознают имена Scanner, Parser и CodeGen, все из которых являются компонентами классической реализации компилятора. Вы можете думать, что я отказался от своих обязательств по отношению к философии KISS и отдрейфовал к более стандартной архитектуре чем мы имели. Более пристальный взгляд, однако, должен убедить вас, что хотя имена схожи, функциональность совершенно различна.

Вместе, сканер и парсер классической реализации составляют так называемый "front end", а генератор кода "back end". Подпрограммы "front end" обрабатывают языкозависимые, связанные с синтаксисом аспекты исходного языка, в то время как генератор кода, или "back end", работает с зависимыми от целевой машины частями проблемы. В классических компиляторах два конца (ends) сообщаются через файл инструкций, написанный на промежуточном языке (IL).

Как правило, классический сканер это одиночная процедура, оперирующая как сопроцедура с синтаксическим анализатором. Она "токенизирует" исходный файл, считывая его символ за символом, распознавая элементы языка, транслируя их в токены и передавая их синтаксическому анализатору. Вы можете думать о синтаксическом анализаторе как об абстрактной машине, выполняющей "op кода", которыми являются токены. Точно также, синтаксический анализатор генерирует "op кода" второй абстрактной машины, которая механизирует IL. Как правило, IL файл записывается на диск синтаксическим анализатором и считывается снова генератором кода.

Наша организация совершенно другая. Мы не имеем лексического анализатора в классическом смысле; наш модуль Scanner, хотя и имеет схожее имя, не является одиночной процедурой или сопроцедурой, а просто набором раздельных подпрограмм, которые вызываются синтаксическим анализатором когда необходимо.

Аналогично, классический генератор кода, "back end", в своем роде тоже транслятор, считывающий "исходный" IL файл и выдающий объектный файл. Наш генератор кода не работает таким способом. В нашем компиляторе нет никакого промежуточного языка; каждая конструкция в синтаксисе исходного языка преобразуется в ассемблер как только она распознана синтаксическим анализатором. Подобно Scanner, модуль CodeGen состоит из индивидуальных процедур, которые вызываются синтаксическим анализатором когда необходимо.

Философия "кодируй как только найдешь" не может производить самый эффективный код в мире - например, мы не обеспечили (пока!) удобное место для оптимизатора - но она несомненно упрощает компилятор, не правда ли?

И этот наблюдение заставляет меня повторить снова то, как нам удавалось сводить функции компилятора к таким сравнительно простым условиям. Я набрался красноречивости на эту тему в прошлых главах, поэтому здесь я не буду слишком ее трогать. Однако, из-за времени, прошедшего с этих последних монологов, я надеюсь что вы предоставите мне совсем немного времени напомнить себе, так же как и вам, как мы попали сюда. Мы дошли до этого применяя несколько принципов, которые создатели коммерческих компилятором редко имеют роскошь использовать. Вот они:

- Философия KISS - никогда не делай сложные вещи без причины.
- Ленивое кодирование - Никогда не откладывай на завтра то, что можешь отложить навсегда. (П. Дж. Плоджер).
- Скептицизм - упрамо отказывайтесь делать что-либо только потому, что это всегда делалось таким способом.
- Принятие неэффективного кода.
- Отклонение произвольных ограничений.

Когда я сделал обзор истории конструирования компиляторов, я узнал, что практически каждый промышленный компилятор в истории страдал из-за предналоженных условий, которые сильно влияли на его дизайн. Первоначальный компилятор Fortran Джона Бэкуса должен был конкурировать с ассемблером и следовательно был вынужден производить чрезвычайно эффективный код. Компиляторы IBM для мини ЭВМ 70-х должны были выполняться в очень небольших объемах ОЗУ тогда доступных - таких небольших как 4k. Ранние компиляторы Ada должны были компилировать себя. Бринч Хансен решил, что его компилятор Паскаля, разработанный для IBM PC должен выполняться на 64k машинах. Компиляторы, разработанные на курсах Computer Science должны были компилировать широкий диапазон языков и следовательно требовали LALR синтаксических анализаторов.

В каждом из этих случаев эти предвзятые ограничения буквально доминировали над проектом компилятора.

Хороший пример - компилятор Бринч Хансена, описанный в его превосходной книге "Brinch Hansen on Pascal Compilers" (строго рекомендую). Хотя его компилятор один из самых ясных и незатемненных реализаций компилятора, что я видел, одно решение, компилировать большие файлы в небольшом ОЗУ, полностью управляло дизайном и он закончил не на одном а многих промежуточных файлах, как и управляющими ими программах для их записи и считывания.

Временами, архитектуры, возникающие из таких решений, находили свое место в учениях компьютерной науки и принимались на веру. По мнению одного человека, пришло время чтобы они были критически пересмотрены. Условия, требования, среды, которые вели к классическим архитектурам не такие же, какие мы имеем сейчас. Нет никакой причины полагать, что решения тоже должны быть те же самыми.

В этой обучающей серии мы следовали по шагам таких пионеров в мире маленьких компиляторов для PC как Леор Золман, Рон Каин и Джеймс Хендрих, тех кто не знал

достаточно теорию компиляции чтобы знать, что они "не могли делать это таким способом". Мы решительно отказались принимать произвольные ограничения, а скорее делали так, как было проще В результате мы развили архитектуру, которая, хотя и совершенно отлична от классической, делает работу простым и прямым способом.

Я закончу эти философствования обзором понятия промежуточного языка. Хотя я отметил перед этим, что мы не имеем его в нашем компиляторе, это не совсем точно; у нас он есть, или по крайней мере мы развиваем его, в том смысле, что мы определяем функции генерации кода для вызова из парсера. В сущности, каждый вызов процедуры генерации кода можно рассматривать как инструкцию на промежуточном языке. Если мы когда либо найдем необходимым формализировать промежуточный язык, вот способ, которым бы мы сделали это: выдать кода из синтаксического анализатора, представляющие собой вызовы процедур генератора кода, а затем обработать каждый код вызывая эти процедуры в отдельном проходе, реализованном в "back end". Откровенно говоря, я не вижу, что мы когда либо найдем потребность в таком подходе, но это связь, если вы решите следовать ему, между классическим и текущим подходами.

РАСШИРЕНИЕ СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА

Хотя я обещал вам где-то в Главе 14, что мы никогда снова не будем переписывать каждую одиночную функцию заново, я начал делать это с Главы 15. Единственная причина: эта длинная пауз между двумя главами делала обзор кажущимся чрезвычайно оправданным... даже необходимым и для вас и для меня. Более важно, решение собрать процедуры в модули заставило нас взглянуть на каждую из них снова, хотели мы этого или нет. И, наконец, откровенно говоря, за последние четыре года у меня появились некоторые новые идеи, которые гарантировали свежий взгляд на некоторые старые вещи. Когда я сперва начал эту серию я был искренне поражен и обрадован, узнав насколько простыми могут быть сделаны подпрограммы анализа. Но в этот последний раз я удивил сам себя снова и был способен делать их точно также, но даже немного проще.

Однако, из-за тотального переписывания модулей синтаксического анализа я не был только способен включить многого в последнюю главу. Из-за этого наш герой, синтаксический анализатор, когда мы последний раз его видели, был только тенью себя прежнего, содержащий только код, достаточный для анализа и обработки показателя состоящего или из переменной или константы. Основным достижением этой текущей главы должно стать восстановление синтаксического анализатора в его прежней славе. В этом процессе, я надеюсь, вы будете терпеливы, если мы иногда будем рассматривать основы, с которые мы имели дело и давно уже прошли.

Сначала, давайте позаботимся о проблеме, к которой мы обращались прежде: наша текущая версия процедуры Factor, как мы оставили ее в Главе 15, не может обрабатывать отрицательные параметры. Чтобы исправить это мы представим процедуру SignedFactor:

```
-- ------------------------------------------------------------
-- Parse and translate a factor with optional sign
: signedfactor ( -- )
        Look >R
        Look addop? IF  getchar  ENDIF
        factor
        R> '-' = IF  _negate  ENDIF ;
-- ------------------------------------------------------------
```

Заметьте, что эта процедура вызывает новую подпрограмму генерации кода Negate:

```
{--------------------------------------------------------------
-- Negate primary
: _negate ( -- ) S" eax neg," emitln ;
-- ------------------------------------------------------------
```

(Здесь и в других местах в этой серии я собираюсь только показывать вам новые подпрограммы. Я рассчитываю, что вы поместите их в соответствующий модуль, который вы должны без проблем определить. Не забывайте добавлять заголовок процедуры в раздел interface модуля.)

В основной программе просто измените вызов процедуры Factor на SignedFactor(main1.frt и протестируйте код. Разве не хорошо компоновщик Turbo и средство make поддерживают все детали?

Да, я знаю, код не очень эффективен. Если мы введем число -3 будет сгенерирован такой код:

```
3 d# -> eax  mov,
eax neg,
```

что действительно, действительно грубо. Мы можем сделать лучше, конечно, просто предварительно добавив знак минус к строке, передаваемой в LoadConstant, но это добавляет несколько строк кода в SignedFactor и здесь я применяю философию KISS очень агрессивно. Более того, сказать правду, я думаю что подсознательно наслаждаюсь генерацией "действительно грубого" кода, так как я могу иметь удовольствие наблюдать как он будет становиться драматически лучше, когда мы примемся за методы оптимизации.

Большинство из вас никогда не слышало о Джоне Спрее, поэтому позвольте мне представить его вам здесь. Джон из Новой Зеландии и преподает информатику в одном из ее университетов. Джон написал компилятор для Motorola 6809, основанный на восхитительном, Паскаль-подобном языке собственной разработки, названном "Whimsical". Позднее он перенес компилятор на 68000 и некоторое время это был единственный компилятор, который я имел для своей доморощенной системы на 68000.

К слову сказать, один из моих стандартных тестов для любого компилятора - изучение того, как компилятор работает с пустой программой типа:

```
program main;
begin
end.
```

Мой тест измеряет время, требуемое на компиляцию и связывание и размер сгенерированного объектного файла. Бесспорный проигравший в этом тесте - компилятор DEC C для VAX, который тратит 60 секунд на компиляцию на VAX 11/780 и генерирует объектный файл 50k. Компилятор Джона бесспорно сейчас, в будущем и навсегда король по части размера кода. Для данной пустой программе Whimsical генерирует точно два байта, реализуя одну инструкцию:

```
RET
```

Устанавливая опцию компилятора генерировать include файл а не автономную программу, Джон может даже урезать этот размер с двух байт до нуля! Несколько трудно добиться нулевого обьектного файла, вы не согласны?

Само собой разумеется, что я рассматриваю Джона как эксперта в оптимизации кода и мне нравится что он однажды сказал: "Лучший способ оптимизации - не оптимизировать вообще, а изначально производить хороший код". Слова, по которым стоит жить. Когда мы начнем оптимизацию мы будем следовать уведомлению Джона и нашим первым шагом будет не добавление щелевого оптимизатора или другого постфактного устройства, но улучшение качества выдаваемого кода перед оптимизацией. Поэтому пометьте SignedFactor как первого хорошего кандидата на внимание и пока оставим его.

ТЕРМЫ И ВЫРАЖЕНИЯ

Я уверен вы знаете, что будет дальше. Мы должны еще раз создать остальные процедуры, которые реализуют синтаксический анализ выражений по методу рекурсивного спуска. Все мы знаем, что иерархия процедур для арифметических выражений такая:

expression

```
        term
                factor
```

Однако сейчас давайте продолжим разработку по шагам и рассмотрим выражения только с аддитивными термами. Код для реализации выражений, включающих возможно первый терм со знаком, показан ниже:

```
-- ---------------------------------------------------------------
-- Parse and translate an expression
: expression ( -- )
        signedfactor
        BEGIN  Look addop?
        WHILE  CASE Look
                 '+' OF  add      ENDOF
                 '-' OF  subtract ENDOF
               ENDCASE
        REPEAT ;
-- ---------------------------------------------------------------
```

Эта процедура вызывает две другие процедуры для обработки операций:

```
-- ---------------------------------------------------------------
-- Parse and translate an addition operation
: add ( -- ) '+' match _push factor _popadd ;
-- ---------------------------------------------------------------
-- Parse and translate a subtraction operation
: subtract ( -- ) '-' match _push factor _popsub ;
-- ---------------------------------------------------------------
```

Эти три процедуры Push, PopAdd и PopSub - новые подпрограммы генерации кода. Как подразумевает имя, процедура Push генерирует код для помещения основного регистра (D0 в нашей реализации для 68000) в стек. PopAdd и PopSub выталкивают вершину стека и прибавляют или вычитают ее из основного регистра. Код показан ниже:

```
-- -----------------------------------------------------------------------
-- Push primary to stack
: _push   ( -- ) S" eax  push,"  emitln ;
-- -----------------------------------------------------------------------
-- Add TOS to primary
: _popadd ( -- ) S" [esp] -> eax  add,  [esp 4 +] -> esp lea," emitln ;
-- -----------------------------------------------------------------------
-- Subtract TOS from primary
: _popsub ( -- ) S" [esp] -> eax  sub,  [esp 4 +] -> esp lea," emitln _negate ;
-- -----------------------------------------------------------------------
```

Добавьте эти подпрограммы в Parser и CodeGen и измените основную программу (main2.frt)для вызова Expression. Вуаля!

Следующий шаг, конечно, это добавление возможности работы с мульпликативными термами. С этой целью мы добавим процедуру Term и процедуры генерации кода PopMul и PopDiv. Эти процедуры генерации кода показаны ниже:

```
-- ------------------------------------------------------------------
-- Multiply TOS and primary
```

```
: _popmul ( -- )
        S" [esp] dword mul, [esp 4 +] -> esp lea," emitln ;
-- ------------------------------------------------------------------
-- Divide primary by TOS
: _popdiv ( -- )
        S" ecx pop, ecx -> eax xchg," emitln
        S" eax -> edx mov, #31 b# -> edx sar, ebx idiv," emitln ;
-- ------------------------------------------------------------------
```

Я должен признать, что подпрограмма деления немного перегружена, но с этим ничего нельзя поделать. К сожалению, хотя процессор 68000 позволяет выполнять деление используя вершину стека (TOS), он требует аргументы в неправильном порядке, подобно тому как для вычитания. Поэтому наше единственное спасение в том чтобы вытолкнуть стек в рабочий регистр (D7), выполнить там деление, и затем поместить результат обратно в наш основной регистр D0. Обратите внимание на использование знаковых операций умножения и деления. Этим неявно подразумевается что все наши переменные будут 16-разрядными целыми числами со знаком. Это решение затронет нас позднее, когда мы начнем рассматривать множественные типы данных, преобразования типов и т.п.

Наша процедура Term это практически аналог Expression и выглядит так:

```
-- ---------------------------------------------------------------
-- Parse and translate a term
: term ( -- )
        factor
        BEGIN   Look mulop?
        WHILE   _push
                CASE Look
                  '*' OF multiply  ENDOF
                  '/' OF divide    ENDOF
                ENDCASE
        REPEAT ;
-- ---------------------------------------------------------------
```

Наш следующий шаг - изменение некоторых имен. SignedFactor теперь становится SignedTerm а вызовы Factor в Expression, Add, Subtract и SignedTerm заменяются на вызов Term:

```
-- ---------------------------------------------------------------
-- Parse and translate a term with optional leading sign
: signedterm ( -- )
        Look >R
        Look addop? IF  getchar  ENDIF
        term
        R> '-' = IF  _negate  ENDIF ;
-- ---------------------------------------------------------------
...
-- ---------------------------------------------------------------
-- Parse and translate an expression
: expression ( -- )
        signedterm
        BEGIN  Look addop?
        WHILE  CASE Look
                 '+' OF  add      ENDOF
                 '-' OF  subtract ENDOF
               ENDCASE
        REPEAT ;
-- ---------------------------------------------------------------
```

Если память мне не изменяет мы однажды уже имели и процедуру SignedFactor и SignedTerm. У меня были причины сделать так в то время... они имели отношение к обработке

булевой алгебры и, в частности, булевой функции "not". Но, конечно, для арифметических операций дублирование не нужно. В выражении типа:

```
-x*y
```

очевидно, что знак идет со всем термом x*y а не просто с показателем x и таким способом Expression и закодирован.

Протестируйте этот новый код, выполнив Main. Она все еще вызывает Expression, так что теперь вы должны быть способны работать с выражениями, содержащими любую из четырех арифметических операций.

Наше последнее дело, относительно выражений, это модификация процедуры Factor для разрешения выражений в скобках. Используя рекурсивный вызов Expression мы можем уменьшить необходимый код практически до нуля. Пять строк, добавленные в Factor, выполнят эту работу:

```
-- ---------------------------------------------------------------
-- Parse and Translate a Factor
: factor ( -- )
        Look '(' = IF '(' match expression ')' match EXIT ENDIF
        Look digit? IF  getnumber loadconstant  EXIT  ENDIF
        Look alpha? IF  getname   loadvariable  EXIT  ENDIF
        S" Unrecognized character `" Look CHAR-APPEND
           &' CHAR-APPEND errors ;
-- ---------------------------------------------------------------
```

К этому моменту ваш "компилятор" должен уметь обрабатывать любые допустимые выражения, которые вы ему подбросите. Еще лучше, что он должен отклонить все недопустимые!

ПРИСВАИВАНИЯ

Пока мы здесь, мы могли бы также написать код для работы с операциями присваивания. Этот код должен только запомнить имя конечной переменной, где мы должны сохранить результат выражения, вызвать Expression, затем сохранить число. Процедура показана дальше:

```
-- ---------------------------------------------------------------
-- Parse and translate an assignment statement
: assignment ( -- )
        getname DUP 1+ ALLOCATE ?ALLOCATE LOCAL name
        name PACK DROP
        '=' match  expression
        name COUNT storevariable
        name FREE ?ALLOCATE ;
-- ---------------------------------------------------------------
```

Присваивание вызывает еще одну подпрограмму генерации кода:

```
-- ---------------------------------------------------------------
-- Store primary register in variable
: storevariable ( c-addr u -- )
        S" eax -> " 2SWAP $+ S"  dword-ptr  mov," $+ emitln ;
-- ---------------------------------------------------------------
```

Теперь измените вызов в Main на вызов Assignment main3.frtи вы должны увидеть полную операцию присваивания, обрабатываемую правильно. Довольно хорошо, не правда ли? И безболезненно также.

В прошлом мы всегда старались показывать БНФ уравнения для определения синтаксиса, который мы разрабатываем. Я не сделал этого здесь и давно пора это сделать. Вот эти БНФ:

```
<factor>      ::= <variable> | <constant> | '(' <expression> ')'
<signed_term> ::= [<addop>] <term>
<term>        ::= <factor> (<mulop> <factor>)*
<expression>  ::= <signed_term> (<addop> <term>)*
<assignment>  ::= <variable> '=' <expression>
```

БУЛЕВА АЛГЕБРА

Следующий шаг, как мы изучили несколько раз до этого, это добавление булевой алгебры. В прошлом этот шаг по крайней мере удваивал количество кода, который мы должны были написать. Когда я прошел эти шаги в своем уме, я обнаружил, что отклоняюсь все больше и больше от того, что мы делали в предыдущих главах. Чтобы освежить вашу память, я отметил, что Паскаль обрабатывает булевы операторы в значительной степени идентично способу, которым он обрабатывает арифметические операторы. Булево "and" имеет тот же самый уровень приоритета, что и умножение, а "or" то же что сложение. Си, с другой стороны, устанавливает их на различных уровнях приоритета, которые занимают 17 уровней. В нашей более ранней работе я выбрал что-то среднее, с семью уровнями. В результате, мы закончили на чем-то называющемся булевыми выражениями, соответствующим в большинстве деталей арифметическим выражениям, но на другом уровне приоритета. Все это, как оказалось, возникло потому, что мне не хотелось помещать скобки вокруг булевых выражений в утверждениях типа:

```
IF (c >= 'A') and (c <= 'Z') then ...
```

При взгляде назад, это кажется довольно мелкой причиной для добавления многих уровней сложности в синтаксический анализатор. Возможно более существенно то, что я не уверен что был даже способен избежать скобок.

Чтобы оттолкнуться, давайте начнем заново, применяя более Паскаль-подобный подход и просто обрабатывая булевы операторы на том же самом уровне приоритетов что и арифметические. Мы увидим, куда это нас приведет. Если это окажется тупиком, мы всегда сможем возвратиться к предыдущему подходу.

Сперва, мы добавим в Expression операторы "уровня сложения". Это легко сделать; во первых, измените функцию IsAddop в модуле Scanner чтобы включить два дополнительных оператора: '|' для "или" и "~" для "исключающее или":

```
-- -----------------------------------------------------------
: addop? ( char -- tf )
        DUP  '+' =
        OVER '-' =  OR
        OVER '|' =  OR
        SWAP '~' =  OR ;
-- -----------------------------------------------------------
```

Затем, мы должны включить анализ операторов в процедуру Expression:

```
-- -----------------------------------------------------------
:NONAME ( -- )
        signedterm
        BEGIN  Look addop?
        WHILE  CASE Look
                 '+' OF  add      ENDOF
                 '-' OF  subtract ENDOF
                 '|' OF  _or      ENDOF
                 '~' OF  _xor     ENDOF
               ENDCASE
        REPEAT ; IS expression
-- -----------------------------------------------------------
```

(Символы подчеркивания необходимы, конечно, потому что "or" and "xor" являются зарезервированными словами Turbo Pascal).

Затем процедуры _Or and _Xor:

```
-- -----------------------------------------------------------------
-- Parse and translate OR
: _or ( -- ) '|' match _push term _popor ;
-- -----------------------------------------------------------------
-- Parse and translate XOR
: _xor ( -- ) '~' match _push term _popxor ;
-- -----------------------------------------------------------------
```

И, наконец, новые процедуры генерации кода:

```
-- -------------------------------------------------------------------------
-- or TOS to primary
: _popor  ( -- ) S" [esp] -> eax  or,   [esp 4 +] -> esp lea," emitln ;
-- -------------------------------------------------------------------------
-- xor TOS to primary
: _popxor ( -- ) S" [esp] -> eax  xor,  [esp 4 +] -> esp lea," emitln ;
-- -------------------------------------------------------------------------
```

Теперь давайте протестируем транслятор (вы возможно захотите изменить вызов в Main обратно на вызов Expression просто чтобы избежать необходимости набирать каждый раз "x=" для присваивания).

Пока все хорошо. Синтаксический анализатор четко обрабатывает выражения вида:

```
x|y~z
```

К сожалению, он также не делает ничего для того, чтобы защитить нас от смешивания булевой и арифметической алгебры. Он радостно сгенерирует код для:

```
(a+b)*(c~d)
```

Мы говорили об этом немного в прошлом. Вообще, правила какие операции допустимы а какие нет не могут быть применены самим синтаксическим анализатором, потому что они не являются частью синтаксиса языка, а скорее его семантики. Компилятор, который не разрешает смешанные выражения такого вида должен распознать, что c и d являются булевыми переменными а не числовыми и передумать об их умножении на следующем шаге. Но такая "охрана" не может быть выполнена синтаксическим анализатором; она должна быть обработана где-то между синтаксическим анализатором и генератором кода. Мы пока не в таком положении, чтобы устанавливать такие правила, потом что у нас нет способа ни объявления типов ни таблицы идентификаторов для сохранения в ней типов. Так что, для того что у нас на данный момент работает, синтаксический анализатор делает точно то, что он предназначен делать.

В любом случае, уверены ли мы, что не хотим разрешить операции над смешанными типами? Некоторое время назад мы приняли решение (или по крайней мере я принял) чтобы принимать значение 0000 как логическую "ложь" и -1 или FFFFh как логическую "истину". Хорошо в этом выборе то, что побитовые операции работают точно таким же способом, что и логические. Другими словами, когда мы выполняем операцию с одним битом логической переменной, мы делаем это над всеми из них. Это означает, что мы не должны делать различия между логическими и поразрядными операциями, как это сделано в C операторами & и &&, и | и ||. Уменьшение числа операторов наполовину конечно не выглядит совсем плохим.

С точки зрения данных в памяти, конечно, компьютер и компилятор не слишком

интересуются представляет ли число FFFFh логическую истину или число -1. Должны ли мы? Я думаю что нет. Я могу придумать множество примеров (хотя они могут быть рассмотрены как "мудреный" код) где возможность смешивать типы могла бы пригодиться. Пример, функция дельты Дирака, которая могла бы быть закодирована в одной простой строке:

```
-(x=0)
```

или функция абсолютного значения (определенно сложный код!):

```
x*(1+2*(x<0))
```

Пожалуйста, заметьте, что я не защищаю программирование подобным образом как стиль жизни. Я почти обязательно написал бы эти функции в более читаемой форме, используя IF, только для того, чтобы защитить от запутывания того, кто будет сопровождать программу в будущем. Все же возникает моральный вопрос: Имеем ли мы право осуществлять наши идеи о хорошей практике кодирования на программисте, написав язык так, чтобы он не смог сделать что-нибудь не так? Это то, что сделал Никлаус Вирт во многих местах Паскаля и Паскаль критиковался за это - как не такой "прощающий" как Си.

Интересная параллель представлена в примере дизайна Motorola 68000. Хотя Motorola громко хвастается об ортогональности их набора инструкций, факт то, что он является далеко не ортогональным. К примеру, вы можете считать переменную по ее адресу:

```
MOVE X,D0 (где X это имя переменной)
```

но вы не можете записать ее таким же образом. Для записи вы должны загрузить в регистр адреса адрес X. То же самое остается истиной и для PC-относительной адресации.

```
MOVE X(PC),DO (допустимо)
MOVE D0,X(PC) (недопустимо)
```

Когда вы начинаете спрашивать, как возникло такое неортогональное поведение, вы находите, что кто-то в Motorola имел некоторые теории о том, как должно писаться программное обеспечение. В частности, в этом случае они решили, что самомодифицирующийся код, который вы можете реализовать, используя PC-относительные записи - Плохая Вещъ. Следовательно, они разработали процессор, запрещающий это. К сожалению, по ходу дела они также запретили все записи в форме, показанной выше, даже полезные. Заметьте, что это было не что-то, сделанное по умолчанию. Должна была быть сделана дополнительная дизайнерская работа, добавлены дополнительные ограничения для уничтожения естественной ортогональности набора инструкций.

Один из уроков, которым я научился в жизни: Если у вас есть два выбора и вы не можете решить которому их них последовать, иногда самое лучшее - не делать ничего. Зачем добавлять дополнительные ограничители в процессор, чтобы осуществить чужие представления о хорошей практике программирования? Оставьте эти инструкции и позвольте программистам поспорить что такое хорошая практика программирования. Точно так же, почему мы должны добавлять дополнительный код в наш синтаксический анализатор для проверки и предупреждения условий, которые пользователь мог бы предпочесть использовать? Я предпочел бы оставить компилятор простым и позволить программным экспертам спорить, должна ли такая практика использоваться или нет.

Все это служит как объяснение моего решения как избежать смешанной арифметики: я не буду ее избегать. Для языка, предназначенного для системного программирования, чем меньше правил, тем лучше. Если вы не согласны, и хотите выполнять проверку на такие условия, мы сможем сделать это, когда у нас будет таблица идентификаторов.

БУЛЕВО "AND"

С это небольшой философией, мы можем приступить к оператору "and", который пойдет в процедуру Term. К настоящему времени вы возможно сможете сделать это без меня, но в любом случае вот код:

В Scanner:

```
-- -------------------------------------------------------------
: mulop?   ( char -- tf )
        DUP  '*' =
        OVER '/' = OR
        SWAP '&' = OR ;
-- -------------------------------------------------------------
```

в Parser:

```
-- -------------------------------------------------------------
-- Parse and translate a term
: term ( -- )
        factor
        BEGIN   Look mulop?
        WHILE   _push
                CASE Look
                  '*' OF multiply  ENDOF
                  '/' OF divide    ENDOF
                  '&' OF _and      ENDOF
                ENDCASE
        REPEAT ;
-- -------------------------------------------------------------
-- Parse and translate AND
: _and ( -- ) '&' match _push factor _popand ;
-- -------------------------------------------------------------
```

и в CodeGen:

```
-- ---------------------------------------------------------------------
-- And TOS to primary
: _popand ( -- ) S" [esp] -> eax  and,  [esp 4 +] -> esp lea," emitln ;
-- ---------------------------------------------------------------------
```

Ваш синтаксический анализатор теперь должен быть способен обрабатывать почти любые виды логических выражений а также (если вы хотите) и смешанные выражения.

Почему не "все виды логических выражений"? Потому что пока мы не имели дела с логическим оператором "not" и с ним все становится сложнее. Логический оператор "not" кажется на первый взгляд идентичным в своем поведении унарному минусу, поэтому моей первой мыслью было позволить оператору исключающего или, '~', дублировать унарный "not". Это не работало. При моей первой попытке процедура SignedTerm просто съедала мой '~' потому что символ проходил проверку на addop но SignedTerm игнорировал все addop за исключением "-". Было бы достаточно просто добавить другую строку в SignedTerm, но это все равно не решит проблему, потому что, заметьте, Expression принимает терм со знаком только для первого аргумента.

Математически, выражение типа:

```
-a * -b
```

имеет небольшой или совсем никакого смысла и синтаксический анализатор должен отметить его как ошибку. Но то же самое выражение, использующее логическое "not", имеет точный смысл:

```
not a and not b
```

В случае с этими унарными операторами выбор заставить их работать таким же самым способом кажется исскуственным принуждением, жертвованием примлемым поведением на алтаре простоты реализуемости. Хотя я полностью за сохранение реализации настолько простой, насколько возможно, я не думаю, что мы должны делать это за счет приемлемости. Исправления подобные этому, приведут к потере основной детали, которая заключается в том, чтобы логическое "not" просто не является тем же самым что унарный минус. Рассмотрим исключающее "or", которое обычно записывается так:

```
a~b ::= (a and not b) or (not a and b)
```

Если мы разрешим "not" изменять весь терм, последний терм в круглых скобках интерпретировался бы как:

```
not(a and b)
```

что совсем не то же самое. Так что ясно, что о логическом "not" нужно думать как о связанном с показателем а не термом.

Идея перегрузки оператор '~' не имеет смысла и с математической точки зрения. Применение унарного минуса эквивалентно вычитанию из нуля:

```
-x <=> 0-x
```

Фактически, в одной из моих более простых версий Expression я реагировал на ведущий addop просто предзагружая нуль, затем обрабатывая оператор как если бы это был двоичный оператор. Но "not" это не эквивалент исключающему или с нулем... которое просто возвратит исходное число. Вместо этого, это исключающее или с FFFFh или -1.

Короче говоря, кажущаяся близость между унарным "not" и унарным минусом разваливается при более близком исследованиии. "not" изменяет показатель а не терм и он не имеет отношения ни к унарному минусу, ни исключающему или. Следовательно, он заслуживает своего собственного символа для вызова. Какой символ лучше, чем очевидный, также используемый в Си символ "!"? Используя правила того как мы думаем должен вести себя "not", мы должны быть способны закодировать исключающее или (предполагая что это нам когда-нибудь понадобится) в очень естественной форме:

```
a & !b | !a & b
```

Обратите внимание, что никаких круглых скобок не требуется - выбранные нам уровни приоритета автоматически заботятся обо всем.

Если вы продолжаете учитывать уровни приоритета, это определение помещает '!' на вершину кучи. Уровни становятся:

1. !
2. - (унарный)
3. *, /, &
4. +, -, |, ~

Рассматривая этот список, конечно не трудно увидеть, почему мы имели проблему при использовании '~' как символа "not"!

Так, как мы механизируем эти правила? Таким же самым способом, как мы сделали с SignedTerm, но на уровне показателя. Мы определим процедуру NotFactor:

```
-- -------------------------------------------------------------
-- Parse and translate a factor with optional "not"
: notfactor ( -- )
        Look '!' <> IF  factor EXIT  ENDIF
```

```
        '!' match  factor _not ;
-- -------------------------------------------------------------
```

и вызовем ее из всех мест, где мы прежде вызывали Factor, т.е. из Term, Multiply, Divide и _And. Обратите внимание на новую процедуру генерации кода:

```
-- -------------------------------------------------------------
-- Bitwise not primary
: _not ( -- ) S" eax not," emitln ;
-- -------------------------------------------------------------
```

Испытайте ее сейчас с несколькими простыми случаями. Фактически, попробуйте пример с исключающим или:

```
        a&!b|!a&b
```

Вы должны получить код (без комментариев, конечно):

```
        A dword-ptr -> eax  mov,                       ; load a
        eax  push,                                      ; push it
        B dword-ptr -> eax  mov,                       ; load b
        eax not,                                        ; not it
        [esp] -> eax  and,  [esp 4 +] -> esp lea,     ; and with a
        eax  push,                                      ; push result
        A dword-ptr -> eax  mov,                       ; load a
        eax not,                                        ; not it
        eax  push,                                      ; push it
        B dword-ptr -> eax  mov,                       ; load b
        [esp] -> eax  and,  [esp 4 +] -> esp lea,     ; and with !a
        [esp] -> eax  or,   [esp 4 +] -> esp lea,     ; or with first term
```

Это точно то, что мы хотели получить. Так что, по крайней мере, и для арифметических и для логических операторов наш новый приоритет и новый, более тонкий синтаксис, поддерживают друг друга. Даже специфическое, но допустимое выражение с ведущим addop:

```
       ~x
```

имеет смысл. SignedTerm игнорирует ведущий '~' как и должно быть, так как выражение эквивалентно:

```
        0~x,
```

что эквивалентно x.

Когда мы взглянем на созданные нами БНФ, мы обнаружим, что наша булева алгебра добавляет теперь только одну дополнительную строку:

```
<not_factor>  ::= [!] <factor>
<factor>      ::= <variable> | <constant> | '(' <expression> ')'
<signed_term> ::= [<addop>] <term>
<term>        ::= <not_factor> (<mulop> <not_factor>)*
<expression>  ::= <signed_term> (<addop> <term>)*
<assignment>  ::= <variable> '=' <expression>
```

Это большое улучшение предыдущих достижений. Будет ли сохраняться наша удача когда мы примемся за операторы отношений? Мы выясним это скоро, но мы должы будем дождаться следующей главы. У нас выдалась подходящая пауза и я хочу выдать эту главу в ваши руки. Уже прошел год с выпуска Главы 15. Я боюсь признаться, что вся эта текущая глава была готова уже давно, за исключением операторов отношений. Но эта информация совсем не дает вам ничего хорошего, сидя на моем жестком диске, и удерживая ее пока пока

операторы отношений не будут сделаны, я не давал ее в ваши руки все это время. Пришло время выдать ее чтобы вы смогли получить из нее что-нибудь ценное. Кроме того, имеется большое количество серъезных философских вопросов, связанных с операторами отношений, и я предпочел бы сохранить их для отдельной главы, где я смог бы сделать это корректно.

Развлекайтесь с новой более тонкой арифметикой и логическим анализом а я скоро увижу вас с отношениями.